Question1 :

In the worse case, how many guesses would it our guessing game take to get the right answer if we had no hints at all? Explain.

The answer is 10 guesses. What we do is pick up the right one of ten numbers and its probabilities is ten.

The worse case is that we use *the simple search,* with each guess we are eliminating only one number. Then we traverse all numbers.

Question2 :

In the worst case, how many guesses does it take to get the right number if we get a hint of "higher or lower" when guessing numbers 1-10 and guess intelligently (always picking in the middle of the remaining set of numbers)?

The answer is 4 guesses.

The total of numbers (from1 to 10) is 10. We always pick in the middle of the remaining set of numbers(binary search) , the log base is 2, and we have to check log 10 elements in the worst case.

$2^3 = 8$; $2^4 = 16$;

The answer of $\log_2(10)$ is a float between 2 and 3. The guesses should be an integer.

So it takes 4 guesses to get the right number.

hw02-exercises

Author’s name: Cuiting Huang

Author affiliation: Northeastern University

City & country location: Zhengzhou, China

E-mail address: huang.cui@northeastern.edu

Abstract – This is an exercises of the Homework - Module2. In this part 3, I will do my written exercises. In this homework, I have built a ring buffer queue and a linked stack. Here are some usage of these.

Index Terms – Queue, Stack, Pointer, Ring buffer.

I. Question One

Circular queues are used quite a bit in operating systems and high performance systems, especially when performance matters. Do a little outside research, and edit this section of the readme answering specifically: *Why is a ring buffer useful and/or when should it be used?*

Answer

First there are several things we must know about the Circular queue. A circular buffer starts out empty and has a set length, and uses FIFO(first in, first out) logic. Also it has a property that when it is full and a subsequent write is performed, then it starts overwriting the oldest data. When there is a lot of process and data need to be run,we can not store all the data. Our computer processes a data and release it, then process the next one. It will cause the memory wasted of the data that has been processed.

Circular buffers can help us solve the problem completely. It will use only one fixed piece of memory. It help us to avoid creating, canceling, and allocating working memory frequently. We could use less memory and do more processes. Static data storage is used more often in the embedded system development than dynamic allocation.

Here is an example I read the materials from the outside sources. We build a system which needs to operate multiple terminals and monitor their health status in real time. We can use the Circular buffers to achieve this. We set up a socket communication between the server and the terminal, so that the server can maintain a long connection with each terminal, which enables us to realize the server can operate the terminal and receive the health status packets of the terminal[1] .

Circular buffers are also a useful construct for situations where data writes and reads occur at different rates: the latest data is always available. If the speed of reading cannot keep up with the speed of writing, the old data will be overwritten by the new data. By using a circular buffer, we can ensure that we are always using the latest data.

[1] https://www.embdded.com/ring-buffer-basics/

II. Question Two

We are going to talk about stacks quite a lot in this course, so it will be important to understand them. Do a little outside research, and edit this section of the readme answering specifically: Why is a stack useful and/or when should it be used?

Answer

A stack has two main principal operations, PUSH and POP, and it uses LIFO(last in first out) logic. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack[2]. There are two storage structures of stack, sequence stack and linked stack.

A linked stack is used to represent a stack to facilitate the insertion and deletion of nodes. When multiple stacks are used simultaneously in the program, a linked stack can not only improve efficiency, but also achieve the purpose of sharing storage space.

Here are some usage scenario of the stack. When we try to call of a subroutine, the address of the next instruction will be stored in the stack then jumping to the subroutine. And when the subroutine is completed and the address will be removed to return to the original program. Calling a recursion routine is very similar with the subroutine. Also we can use a stack to make a traversal of binary trees and Compute an expression. The stack is used in the second and third part of the compiler's front end called the semantic analyzer and the syntax analyzer.

[2] https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Homework 7 - Analysis

In this homework, we are going to work to become comfortable with the mathematical notation used in algorithmic analysis.

Problem 1: Quantifiers

For each of the following, write an equivalent *English statement.* Then decide whether those statements are true if x and y are integers (e.g., they can be any integer). Then write a convincing argument to prove your claim.

1. $\forall x \, \exists y : x + y = 0$

   English Statement
   For all x ,there exists some y making $\mathrm{x} + \mathrm{y} = 0$ true.

   Decide
   This statement is True.

   Argument
   Every integer has its negative number. There always has a $\mathrm{y} = -x$

2. $\exists y \, \forall x : x + y = x$

   English Statement
   There exists a value of y making $x + y = x$ always true whatever the value of x .

   Decide
   This statement is True.

   Argument
   While y = 0, x could be any integer $0 + x = x$.

3. $\exists x \, \forall y : x + y = x$

   English Statement
   There is a x making $x + y = x$ always true whatever the value of y.

   Decide
   This statement is False.

   Argument
   The statement $x + y = x$ equals to $y = x - x = 0$, which means y can not be any integer. This statement can be true only when y is zero.

Name: Cuiting Huang Homework 7 - Analysis

Problem 2: Growth of Functions

Organize the following functions into six (6) columns. Items in the same column should have the same asymptotic growth rates (they are big-Oh and big-θ of each other. If a column is to the left of another column, all of its growth rates should be slower than those of the column to its right.

$$n^2,\ n!, n\,log_2\,n, 3n, 5n^2+3, 2^n, 10000, n\,log_3\,n, 100, 100n$$

100	$3n$	$n\,log_2\,n$	n^2	2^n	$n!$
10000	$100n$	$n\,log_3\,n$	$5n^2+3$		

Problem 3: Function Growth Language

Match the following English explanations to the *best* corresponding big-Oh function by drawing a line from an element in the left column to an element in the right column.

$$\begin{tikzcd}
\text{Constant} \arrow[rd] & O(n^3) \\
\text{Logarithmic} \arrow[rdd] & O(1) \\
\text{Linear} \arrow[r] & O(n) \\
\text{Quadratic} \arrow[rd] & O(log_2\,n) \\
\text{Cubic} \arrow[ruuuu] & O(n^2) \\
\text{Exponential} \arrow[rd] & O(n!) \\
\text{Factorial} \arrow[ru] & O(2^n)
\end{tikzcd}$$

Problem 4: Big-Oh

Formal Definition
$f(n)$ is $O(g(n))$ if there exists constants 'c' and 'k' such that
$|f(n)| \leq c\ |g(n)|$ wherever $n > k$

1. Using the definition of big-Oh, show that $100n + 5 \in O(2n)$

   Show
   $100n + 5 \in O(2n)$
   $100n + 5\ = \ \leq 50 * 2\ n + 5 * 2n,\ n > 1;$
   $\Rightarrow 100n + 5 \leq 55 * 2n, n > 1;$

   While $n\ =\ 2$, $(100 * 2 + 5\ \leq 105 * 2)$ is true
   $c = 55, k\ = 1$
   $100n + 5 \in O(2n)$ is true

2. Using the definition of big-Oh, show that $n^3 + n^2 + n + 42 \in O(n^3)$

   Show
   $n^3 + n^2 + n + 42 \in O(n^3)$
   $n^3 + n^2 + n + 42\ \leq n^3 + n^3 + n^3 + 42n^3,\ n > 1$
   $\Rightarrow n^3 + n^2 + n + 42 \leq 45n^3$

   While $n = 2$, $(8 + 4 + 2 + 42\ \leq\ 45 *\ 8)$ is true
   $c = 45, k\ = 1$
   $n^3 + n^2 + n + 42 \in O(n^3)$ is true.

3. Using the definition of big-Oh, show that $n^{42} + 1{,}000{,}000 \in O(n^{42})$

   Show
   $n^{42} + 1{,}000{,}000 \in O(n^{42})$
   $n^{42} + 1{,}000{,}000 \leq n^{42} + 1{,}000{,}000n^{42},\ n > 1$
   $\Rightarrow n^{42} + 1{,}000{,}000 \leq 1{,}000{,}001n^{42}$

   While $n = 2$, $(2^{42} + 1{,}000{,}000 \leq 1{,}000{,}001 * 2^{42}\ )$ is true
   $c = 1000001, k\ = 1$
   $n^{42} + 1{,}000{,}000 \in O(n^{42})$ is true.

Name: Cuiting Huang Homework 7 - Analysis

Problem 5: Searching

In this problem, we consider the problem of searching in ordered and unordered arrays:

1. We are given an algorithm called *search* that can tell us *true* or *false* in one step per search query if we have found our desired element in an unordered array of length 2048. How many steps does it takes in the worst possible case to search for a given element in the unordered array?

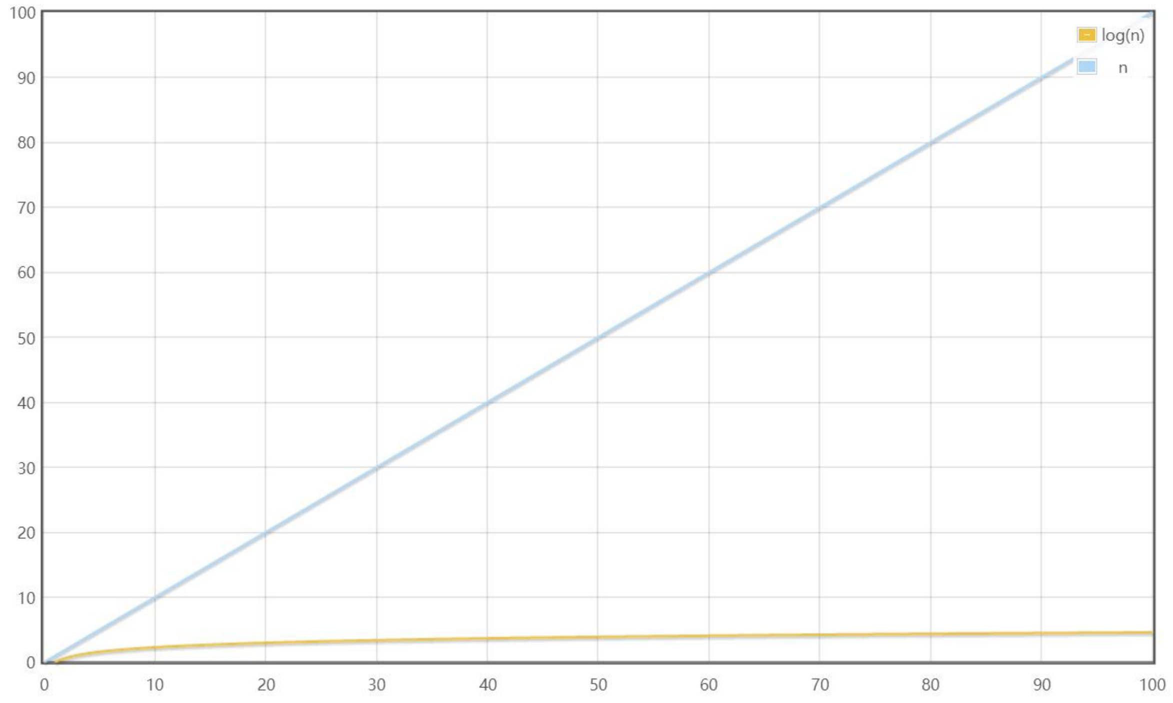
   In the worst possible case, it takes 2048 times to search for a given element in the unordered array.

   We use the *simple search(linear search)*, for each time we get a wrong element, and we just traverse all elements.

2. Describe a *fasterSearch* algorithm to search for an element in an ordered array. In your explanation, include the time complexity using big-Oh notation and draw or otherwise clearly explain why this algorithm is able to run faster.

   In an ordered array, we will use binary search. The big-Oh is O(logn).
   Step one, we will pick up the element at the middle of the array, named A.
   Step two,comparing the given element named N with A, we will get a information of N,bigger of smaller than A.
   Step three, we will pick another element in the middle of the remaining set of elements.
   Then we will repeat Step three until we find the given elements.

   The big-Oh of *linear search* is O(n), The big-Oh of *binary search* is O(logn). It is obvious that binary search in an ordered array run more faster.



3. How many steps does your *fasterSearch* algorithm (from the previous part) take to find an element in an ordered array of length 2,097,152 in the worst case? Show the math to support your claim.

   The total of elements is 2097152, in the worst case we have to check log(2097152) elements.

   $2^{10} = 1024$, 2^{20}=1048576, 2^{21} = 2097152
   log(2097152) = 21.

   So it takes 21 steps when we use the faster Search algorithm.

Name: Cuiting Huang Homework 7 - Analysis

Problem 6: Another Search Analysis

Imagine it is your lucky day, and you are given 100 golden coins. Unfortunately, 99 of the gold coins are fake. The fake gold coins all weight 1 oz. but the real gold weighs 1.0000001 oz. You are also given one balancing scale that can precisely weight each of the two sides. If one side is heavier than the other the other side, you will see the scale tip.

1. Describe an algorithm for finding the real coin. You must also include the algorithm's time complexity. **Hint:** Think carefully – or do this experiment with a roommate and think about how many ways you can prune the maximum number of fake coins using your scale.

   Step one, divide the coins into two parts, each part has 50 coins and put them on the balancing scale, take the heavier side and go next step.

   Step two, divide the heavier side it into two parts again, each part has 25 coins, take the heavier side and go next step.

   Step three, put 1 coin aside and divide the rest 24 coins in to two parts, each part have 12 coins. If both sides of the scale have the same weight. The coin we put aside is the golden coin. Otherwise, take the heavier side and go next step.

   Step four, repeat the step two and each part has 6 coins.

   Step five, repeat the step two and each part has 3 coins.

   Step six, now we've got 3 coins in total. Put a coin on each side of the balancing scale, if both sides of the scale have the same weight, the remaining coin is the golden one. Other wise, the coin which is heavier is the golden coin.

   The algorithm's time complexity is $O(logn)$.

2. How many weightings must you do to find the real coin given your algorithm?

   In the worst case it takes 6 times to find the real golden coin.

   And in the best case it takes 3 times.

Name: Cuiting Huang Homework 7 - Analysis

Problem 7 – Insertion Sort

1. Explain what you think the worst case, big-Oh complexity and the best-case, big-Oh complexity of insertion sort is. Why do you think that?

   Best-case: We have a sequential array that all the elements are already sorted. And the big-Oh complexity of this case is $O(n)$.

   For every elements have already been sorted, we will never walk through the while loop, We only need to compare $(n - 1)$ times to confirm that all data have been sorted. So the big-Oh complexity $O(n)$.

   Worst-case: We have a reversed array that every single element needs to be sorted. And the big-Oh complexity of this case is $O(n^2)$.

   we could only know which element is the smallest after the iteration of the end of the unsorted portion, we must also iterate until the end.

   For one element, we conduct function find Minimum for one time, which takes n steps. For every elements, it needs n*n steps.

2. Do you think that you could have gotten a better big-Oh complexity if you had been able to use additional storage (i.e., your implementation was not *in-place*)?

   I do not thinks so.
   The big-Oh complexity refers to the amount of computational effort required to perform the algorithm, it has no relation with the storage we could use.

   Even if we could be able to use additional storage, which means we could put every element sorted by every loop time in new space. We still need to iterate the array, the big-Oh complexity will not be changed.

Homework 5 - Proofs

In this homework, we're going to work through a few CLRS problems, focusing on the basic searches, sorts, and loop invariant problems we've seen so far. Do your best, write answers that are complete sentences, and make sure that you're using the specific notation and pseudocode styles used in CLRS.

Problem 1: Bubblesort

Suppose that Bubblesort on an array A uses the following pseudocode:

```
1 for i = 1 to A.length – 1
2     for j = A.length downto i + 1
3         if A[j] < A[j-1]
4             exchange A[j] with A[j-1]
```

a. Let A' be the output of Bubblesort on A. To prove that Bubblesort is correct, we need to prove that bubblesort terminates, and that

   A'[1] ≤ A'[2] ≤ … ≤ A'[n] ,

   Where n = A.length. In order to short that Bubblesort actually sorts, what else do we need to prove?

   ANS

   In addition to proving the inequality above, we also need to prove that the elements of *A'* *(output)* are also the elements in *A(input).*

   But since Bubblesort is an algorithm that sorts in-place, I think that proving the inequality only is *enough.*

b. *State precisely a loop invariant for the for loop in lines 2-4 in the pseudocode above, and prove* that this loop invariant is maintained for the entire algorithm. Use the CLRS loop invariant proof in Chapter 2 as a model for this proof.

Invariant: *A*[*j*] is the smallest value in range of *A*[*j ...A.length*]

Initialization:Before the first iteration of the loop, *j =A.length and t*he subarray *A* [*j*] only has one elements. *A*[*j*] is the smallest.

Maintenance: For each iteration of the loop, we compare *A*[*j*] with *A*[*j-1*] and put the smaller element at the position *A*[*j-1*]. At the beginning of next iteration, new *j = j -1* of the last iteration with the smaller element, and all these show that each iteration maintains the loop invariant.

Termination: While *j < i + 1* which means *j = i,* we terminate the loop. In the last iteration while *j= i + 1,* we've already compared *A*[*j*] and *A*[*j-1*]*(which equals to A[i] and A[i+1])* and put the smaller element at the position *j (which is also i).*All value in *A [1...A.length],* the algorithm is correct.

c. Using the termination condition of the loop invariant that you proved in part (b), state a loop invariant for the *for* lop in lines 1-4 that will allow you to prove the inequality in (a). Use the CLRS loop invariant proof in Chapter 2 as a model for this proof.

Invariant: The subarray *A* [*1...i-1*] *is sorted. A*[*i-1*] is the smallest value in range of

A [*i...A.length*]

Initialization: Before the first iteration of the loop*, i =1 and t*he subarray *A* [*1...i -1*] is empty.

Maintenance: Before the i times iteration of the loop, the subarray *A* [*1...i-1*] *is sorted and A*[*i-1*] is the smallest value in range of *A* [*i...A.length*]. In this iteration, according to the answer of b, while we terminate the inner loop*(which means j = i)*, *A*[*i*] is the smallest value in range of *A* [*i+1...A.length*].

Thus, the subarray *A* [*1...i*] *is sorted and A*[*i*] is the smallest value in range of *A* [*i+1...A.length*].

Termination: While *i > A.length - 1* which means *i = A.length,* we terminate the loop. All value in *A* [*1...A.length -1*] are sorted and less than the value of *A.length.*By definition, the values in *A[1...A.length]* are sorted.Hence, the algorithm is correct.

d. What is the worst-case running time of bubblesort? How does it compare to the worst-case running time of insertion sort?
The worst-case is that we have a reverse array need to sort and we need to run the fourth statement for every time.

Statement	Cost	Times
for i = 1 to A.length – 1	c_1	n
for j = A.length downto i + 1	c_2	$\sum_{j=2}^{n} j$
if A[j] < A[j-1]	c_3	$\sum_{j=2}^{n} (j-1)$
exchange A[j] with A[j-1]	c_4	$\sum_{j=2}^{n} (j-1)$

$T(n) = c_1 n + c_2 \sum_{j=2}^{n} j + c_3 \sum_{j=2}^{n} (j-1) + c_4 \sum_{j=2}^{n} (j-1)$
The big-Oh complexity is $O(n^2)$.

Comparing to the worst-case running time of Insertion sort.

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n-1)$$

In the worst-case, they have the same big-Oh complexity $O(n^2)$.

Problem 2: Inversions

Let A[1…n] be an array of *n* distinct numbers (“distinct” means “no duplicates”). If i < j and A[i] > A[j], then the pair (i,j) is called an inversion of A.

a. List the five inversions of the array ⟨ 2, 3, 8, 6, 1 ⟩

   $(1,5),(2,5)(3,5)(4,5)$
   $(3,4)$

b. Imagine an array of length *n*, filled only with integers. For an *n* length array, what does the array look like with the most inversions? How many inversions does it have, written in terms of its length *n*?

   The array is a reverse array. *⟨n, n-1, … 1 ⟩.*
   The number of inversions of the array is
   $$(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}=c_n^2$$

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Give reasons for your answer.

   When there is an inversion, the body of ***the while loop*** of insertion sort will operate once.
   The running time of the while loop in insertion sort is $O(n+m)$, m is the number of inversions in the input array.

Name: Cuiting Huang Homework 05 - Proofs

Problem 3: Binary Search

Refer back to the pseudocode you wrote for linear search in the lab. Let’s change the problem: let’s say that the array A is sorted, and we can check the midpoint of the sequence against *v* and eliminate half of the sequence from our search. The *binary search* algorithm repeats this procedure, halving the size of the remaining sequence every time.

a. Write pseudocode that uses a for or while loop for binary search.
ANS

```
BINARY-SEARCH(A)
int min = 1
int max = A.length
int value v
while(min<=max)
    mid=(min+max)/2
    if A[mid]=value v then
        return mid
    else if A[mid]>value v then
        max=mid-1
    else if A[mid] <value v then
        min=mid+1
return -1 // means we find nothing
```

b. Make an argument based on this pseudocode for the worst-case runtime.
ANS
The worst case:
There is a sequence array *A*[*1....n*]
max = n ,min =1, value v, compare A[*mid*] with value, the rest numbers of array is *n/2,*
max = n/2, min = 1 compare again, the rest numbers of array is *n/4,*
max = n/4, min = 1 compare again, the rest numbers of array is *n/16,*
....
every time, the rest numbers of array is n/(2^m). When n/(2^m) = 1, we could get the *value v* or nothing.
Thus, *n =* (2^m)*,*the worst case runtime of binary search is *m* = $log_2(n)$. The big-Oh is *O(logn).*

c. Prove that binary search solves the search problem on a sorted array by using a loop invariant proof.

Invariant: There is a sequence array *A*[*i....j*], in the middle of array, *mid =(i+j)/2*, A[mid] is not equals to the target value *v.*

Initialization: Before the first iteration of the loop*,*we try to search the target value from *i* to *j* of the array. *i = 1* , *j =* the number of array elements.

Maintenance: For each iteration of the loop, we check *A*[*mid*] is not the target number, and get the new value of *i* and *j.* if *A*[*mid*] > value *k, i = i, j = mid - 1;* if *A*[*mid*] < value *k, i = mid + 1, j = j, which is* showing that each iteration maintains the loop invariant.

Termination: If we terminate the loop with returning index *mid*, it is when we already find an element *A*[*mid*] equals to the value *v.* In this case, our algorithm is correct.

If we terminate the loop without returning index *mid*, which means there is not an element of *A*[*i...j*] equals to the value *k.* And we return -1. Hence, the algorithm is correct.

Name: Cuiting Huang Homework 05 - Proofs

Problem 4: Improve Bubblesort performance

a. Modify the pseudocode of Bubblesort to increase its efficiency marginally.

ANS

```
1 for i = 1 to A.length – 1
2         int flag = 1 //set a flag
3         for j = A.length downto i + 1
4                 if A[j] < A[j-1]
5                         exchange A[j] with A[j-1]
6                         flag = 0 // if we make an exchange, give a new value to flag
7         if flag == 1
8                 break   // if the array is sorted  break the whole loop.
```

b. Write a proof showing that this modification still solves the sorting problem, using loop invariant.

Invariant: The subarray *A* [*1...i-1*] *is sorted. A*[*i-1*] is the smallest value in range of

A [*i...A.length*]

Initialization: Before the first iteration of the loop*, i =1 and t*he subarray *A* [*1...i -1*] is empty.

Maintenance: Before the i times iteration of the loop, the subarray *A* [*1...i-1*] *is sorted and A*[*i-1*] is the smallest value in range of *A* [*i...A.length*].

In this iteration, in the inner for loop, we compare *A*[*j*] with *A*[*j-1*] ,if *A*[*j*] is the smaller one we swap the two numbers and give the flag new value, for each iteration of inner loop, we traverse all element in the array.

While we terminate the inner loop*(which means j = i)*, *A*[*i*] is the smallest value in range of *A* [*i+1...A.length*].

Thus, the subarray *A* [*1...i*] *is sorted and A*[*i*] is the smallest value in range of *A* [*i+1...A.length*].

Termination: if *i <A.length,* we terminate the loop with flag = 1, which means for all loop, All value in *A*[*1...A.length -1*] are sorted, *A*[*i*] is always bigger than *A*[*i-1*] and less than the value of *A.length.*By definition, the values in *A[1...A.length]* are sorted. Hence, the algorithm is correct.

While *i > A.length - 1* which means *i = A.length,* we also terminate the loop. For all loop, *j=i.* All value in *A* [*1...A.length -1*] are sorted and less than the value of *A.length.*By definition, the values in *A* [*1...A.length*] are sorted.Hence, the algorithm is correct.

Name: Cuiting Huang Homework 05 - Proofs

c. Prove that this algorithm is still $O(n^2)$

Statement	Cost	Times
for i = 1 to A.length – 1	c_1	n
int flag = 1	c_2	n
for j = A.length downto i + 1	c_3	$\sum_{j=2}^{n} j$
if A[j] < A[j-1]	c_4	$\sum_{j=2}^{n} (j-1)$
exchange A[j] with A[j-1]	c_5	$\sum_{j=2}^{n} (j-1)$
flag = 0	c_6	$\sum_{j=2}^{n} (j-1)$
if flag = 1	c_7	n

$$T(n) = c_{1n} + c_2 n + c_3 \sum_{j=2}^{n} j + c_4 \sum_{j=2}^{n} (j-1) + c_5 \sum_{j=2}^{n} (j-1) + c_6 \sum_{j=2}^{n} (j-1) + c_7 n$$

$$T(n) = c_1 n + c_2 n + n * c_3(n-1) + n * (c_4 + c_5 + c_6)(n-3) + c_7 n$$

In the best case, we have an sequence array, c3,c4 will be operated for one time and c5.c6 will never be operated. We operate c_7 for once and break the loop.
$T(n) = c_1 n + c_2 n + c_3(n-1) + c_4(n-3) + c_7 n$
$T(n) = (c1 + c_2 + c_3 + c_4 + c_7)n - c_3 - 3c_4$
The big-Oh complexity is $O(n)$.

In the other cases, every statement will be operate at least twice
$T(n) = (c_3 + c_4 + c_5 + c_6)n^2 + (c1 + c_2 - c_3 - 3c_4 - 3c_5 - 3c_6 + c_7)n$
The big-Oh complexity is still $O(n^2)$.

Homework 6 - Brute force, divide and conquer, analysis

Problem 1:

Given an array of size 16, what is the maximum possible *mixed-up-ness* score? Explain why you think this (e.g., give a logical argument or provide an example)

ANS

The maximum possible mixed-up-ness score is 120.

For insertion sort, when we have a reverse array, in the body of the while loop, we need operate the swap statement every time, which meas we will always get a pair of inversion number.
Pseudocode:
INSERTIONSORT(A)

```
1   for i from 1 to size
2   j = i;
3   while j > 0
4       if (array[j] < array[j - 1]) then
5           swap(&array[j], &array[j - 1]);
```

A reverse array.〈 n, n-1, ... 1 〉.

The number of inversions of the array is $(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}=c_n^2$

An Instance array A[6,5,4,3,2,1,0], the number N of array = 7
For A[0]=6, can be consisted of the count inversions with the rest elements of A[5,4,3,2,1,0], we get 6 pairs, which is (n-1)
For A[1]=5, can be consisted of the count inversions with the rest elements of A[4,3,2,1,0], we get 5 pairs,which is (n-2)
For A[2]=4, can be consisted of the count inversions with the rest elements of A[3,2,1,0], we get 4 pairs,which is (n-3)
....
.....
For A[5]=1, can be consisted of the count inversions with the rest elements of A[4,3,2,1,0], we get 1 pairs,which is (n-6)
For A[6]=0, has no count inversion.
Thus, given an array of size 16, the maximum possible mixed-up-ness score is 120 when array is reverse.

Problem 2:

What is the worst-case runtime of the brute-force algorithm that you implemented? Give a proof (a convincing argument) of this.

ANS

Pseudocode:

SLOW MIXED-UP NESS(A, size)

```
1   count equals to 0;
2   i from 0 to size
3       j from i to size
4           if (array[i] > array[j]) then
5               count++;
6   return count;
```

The worst-case runtime of the brute-force algorithm is that we need to run line 5 for every circulation. That means every element A[i] of array is bigger than A[j], which j > i.

The array is a reverse array. [*n, n-1, ... 1*].
For i = 0, A[0] = n, we run line 4 to compare A[0] and A[j],
A[0] is always smaller than A[j], j∈[1.....n-1], we always need to run line 5.
For i = 1, A[1] = n-1, we run line 4 to compare A[1] and A[j],
A[0] is always smaller than A[j], j∈[2.....n-1], we always need to run line 5,
......
....
For i = k, A[k] = n-k, we run line 4 to compare A[1] and A[j],
A[0] is always smaller than A[j], j∈[k+1.....n-1], we always need to run line 5.
The big-Oh is $O(n^2)$

Problem 3:

State the recurrence that results from the divide-and-conquer algorithm you implemented in Part 3.

ANS

Pseudocode:

mergeSort(int arr[], int temp[], int left, int right)

```
1   if (left < right)
2       int mid = (left + right) / 2;
3       mergeSort(arr, temp, left, mid)
4       mergeSort(arr, temp, mid + 1, right)
5       merge(arr, temp, l, mid, right)
```

For this part of code, we have two arrays, the left position of the array which equals to zero, and the right position of the array which equals to size of array minus one.

when left is smaller than right, we get the middle position *mid = (left+right)/2*

First Recursion,we operate the mergeSort in line 3from left to middle, we've got the subarray A[left.....middle], which has half number of A[left...right]. The next we will be back to if statement compare left with new right which is the middle. Then we will do this step recursive until the left equals to middle, which means we get a subarray only has one element A[left].

Second Recursion, we operate the mergeSort in line 4 from mid+1 to right. The last time we run line 4, we put the mid(right) = 1, get mid = 0 =left. For now mid +1 = 1,right =1. We get back to if statement *mid+1 = right* =1. Then we operate the line 5 merge function.

Through all the steps above, we get the subarray with only one element. When we done merge function, we will operate the mergeSort again.

Problem 4:

Solve the recurrence for the divide-and-conquer algorithm using the substitution method. For full credit, show your work.

ANS

Assuming that dividing the array and combing the array take in total cn time, and the total time required by merge sort is $\mathrm{T(n)}$. For each division, we are actually halving the array, and for each smaller array, it needs $\frac{1}{2}\mathrm{T}(\frac{\mathrm{n}}{2})$ time.Then we have

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

$$= 2(2T\left(\frac{n}{4}\right) + \frac{cn}{2}) + cn$$

$$= 2\left[2\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + \frac{cn}{2}\right] + cn$$

$$\ldots \quad = 2^kT\left(\frac{n}{2^k}\right) + kcn$$

Let $n = 2^k, k = \ log_2n$

$T(n) = nT(1) + cnlog_2n = n + cnlog_2n$

$T(n) \in O(nlogn)$

Therefore, we conclude that both the worst-case and best-case big-Oh complexity of merge sort are $O(nlogn)$.

Problem 5:

Confirm that your solution to #4 is correct by solving the recurrence for the divide-and-conquer algorithm using the master theorem. For full credit, clearly define the values of a, b, and d.

ANS

The definition of the master theorem is:

If $T（n） = a * T(\frac{n}{b}) + n^d$ *for a>1, b>1, d≥0*

There are three cases:

if $d > log_b a \quad T(n) = O(n^d)$

else if $d = log_b a \quad T(n) = O(n^d logn)$

else if $d < log_b a \quad T(n) = O(n^{log_b a})$

According to #4, we have the running time of merge sort is $T（n） = 2^n * T(\frac{n}{n^n}) + O(n^1) + O(1)$

which means *a* = 2^n, *b* = 2^n, *d = 1*

According to the Master Theorem, it's the second case,

d = $log_b a = 1, \ T(n) = O(n^d logn) = O(nlogn)$

The big-Oh is the same as the complexity of divide-and-conquer algorithm with recursion, thus it's definitely correct.

Homework 7 - Binary Search Trees

Problem 1:

What does it mean if a binary search tree is a *balanced tree*?

ANS

It means that in this binary tree the depth of the two subtrees of every node never differ by more than one. The binary search tree is height-balanced.

Problem 2:

What is the big-Oh search time for a balanced binary tree? Give a logical argument for your response. You may assume that the binary tree is perfectly balanced and full.

ANS

The big-Oh search time for a balanced binary tree is $O(logn)$.It is similar with the binary search.

Assuming the number of nodes of a balanced binary tree is n, the height of tree is h. The binary tree is perfectly balanced and full, we could get that $n = 2^h - 1$, which also means $h = log(n+1)$. *T(n)* is the average time of we search one target value in a balanced tree.

When $h = 2, n = 3, T(n) = (1*1 + 2*2)/3$

When $h = 3, n = 7\ T(n) = (1*1 + 2*2 + 3*4)/7$

When $\boldsymbol{h = h, n = n,\ T(n) = (1*1 + 2*2 + 3*4 + ... + h*2^{(h-1)})}$/n ①

Then we could get,

$1*1 + 2*2 + 3*4 + ... + h*2^{(h-1)}$ = $1*2^0 + 2*2^1 + 3*2^2 + ... + h*2^{(h-1)}$ **= SUM** ②

$2*\text{SUM} = 2*(1*2^0 + 2*2^1 + 3*2^2 + ... + h*2^{(h-1)})$

=$1*2^1 + 2*2^2 + 3*2^3 + ... + (h-1)*2^{(h-1)} + h*2^h$ ③

Then, we use ③ − ②

SUM = $h*2^h - (2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{(h-1)})$

$\boldsymbol{A} = 2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{(h-1)}$ is the sum of geometric sequence.

Then, $\mathbf{A = 2^h - 1},\ \boldsymbol{h = log(n+1)}$

SUM = $h*2^h - (2^h - 1) = (h-1)*2^h + 1 = [log(n+1) - 1]*2^{log(n+1)} + 1$

$= [log(n+1) - 1]*(n+1) + 1$

$= log(n+1)*(n+1) - n$

So, $\boldsymbol{T(n) = SUM/n = \frac{n+1}{n}log(n+1) - 1}$

The big-Oh search time for a balanced binary tree is $\boldsymbol{O(logn)}$.

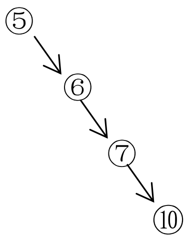
Problem 3:

Now think about a binary search tree in general, and that it is basically a linked list with up to two paths from each node. Could a binary tree ever exhibit an O(n) worst-case search time? Explain why or why not. It may be helpful to think of an example of operations that could exhibit worst-case behavior if you believe it is so.

ANS

The big-Oh search time for a binary tree could be *O(n).*

The worst-case is the binary tree is totally unbalanced, it only has a left path or a right path, which means that we can think of the tree as a linked list. The big-Oh search time is same as a linked list.



In the worst case, we need to traverse all nodes of the unbalanced tree, find the target value or not.

Problem 4:

What is the recurrence relation for a binary search? Your answer should be in the form of $T(n) = aT(n/b) + f(n)$. Clearly state the values for *a*, *b* and *f(n)*.

ANS

Pseudocode:

```
BinarySearch(array A[], int value, int first, int last)
1    if( first > last)
2        return -1// not found target value
3    int mid = ( first + last) / 2;
4    if ( value == A[mid]) then
5        return mid;
6    else
7        if( value  <  A[mid])
8            BinarySearch(A, value, first, mid - 1);
9        else if( value  >  A[mid])
10           BinarySearch(A, value, mid + 1, last);
```

Assuming *T(n)* is the time we spend when we use binary search and the number of array is n.

When n = 1, the algorithm executes from line1 to line5, It takes a "constant" time, T(1) = 1.

When n > 1, the algorithm will execute on line8 or line 10. For both line, they run with the subarray which has $\frac{n}{2}$ elements. The sum of running time from line6 to line 10 is always $\mathrm{T}(\frac{n}{2})$.

Then we have,

$$\begin{cases} T(1) = 1, & n = 1 \\ T(n) = T(\frac{n}{2}) + T(1), & n > 1 \end{cases}$$

So, $T(n) = aT(n/b) + f(n)$ = $T(\frac{n}{2}) + T(1), n > 1.$ $\mathbf{a = 1, b = 2\ f(n) = 1}$

Problem 5:

Solve the recurrence for binary search algorithm using the *substitution method*. For full credit, show your work.

ANS

According to #4, we have

$$\begin{cases} T(1) = 1, & n = 1 \\ T(n) = T(\frac{n}{2}) + T(1), & n > 1 \end{cases}$$

$T(1) = 1,$

$T(2) = 1 + 1 = log_2 2 + 1$

$T(4) = 2 + 1 = log_2 4 + 1$

$T(8) = 3 + 1 = log_2 8 + 1$

$T(16) = 4 + 1 = log_2 16 + 1$

We can deduce that $T(n) = logn + 1$, the big-Oh is $O(logn)$.

Problem 6:

Confirm that your solution to #5 is correct by solving the recurrence for binary search using the *master theorem*. For full credit, clearly define the values of *a*, *b*, and *d*.

ANS

The definition of the master theorem is:

If $T(n) = a * T(\frac{n}{b}) + n^d$ *for a>1, b>1, d≥0*

There are three cases:

if $d > log_b a$ $T(n) = O(n^d)$ ①

else if $d = log_b a$ $T(n) = O(n^d logn)$ ②

else if $d < log_b a$ $T(n) = O(n^{log_b a})$ ③

According to #5, we have

$$\begin{cases} T(1) = 1, & n = 1 \\ T(n) = T(\frac{n}{2}) + T(1), & n > 1 \end{cases}$$

when n > 1, we can know that *a =1*, *b* = 2, *d = 0*

According to the Master Theorem, it's the second case,

$d = log_b a = 0,\ T(n) = O(n^d logn) = O(logn)$

The big-Oh is the same as the complexity of binary search algorithm with recursion, thus it's definitely correct.

Homework 8 - Graphs

Problem 1:

What is the big-Oh space complexity of an adjacency list? Justify your answer.

ANS

Assume that the number of vertices of a graph is ***n**,* and there are ***e*** edges in it. For an adjacency list, the big-Oh space complexity is $O(n + e)$.

We use the adjacency list to represent a graph, which means each node in the graph are stored in a Double Linked List, and for each node storing a list of its neighbors
First for each node of graph, the big-Oh space complexity is $O(n)$. Then,
for undirected graph every edge is stored in DLL for twice, while for directed graph every edge is stored in DLL for once.
The largest number of edges is *2E, which* $\Rightarrow E.$ Thus, for a graph representing by an adjacency list, the big-Oh space complexity is $O(n + e)$.

Problem 2:

What is the big-Oh space complexity of an adjacency matrix? Justify your answer.

ANS

For each vertex, it takes n blocks of space, and its adjacency matrix takes n^2 blocks of space to represent all of the relationships given.
Therefore, given n nodes, the total space required by the adjacency matrix is $n^2 + n$.
The space complexity in Big-O is $O(n^2)$.

Notice that undirected adjacency matrix is symmetrical, one of its diagonals from left top to right bottom is always zero, so it takes $\frac{n(n-1)}{2}$ blocks of space to represent all of the relationships given.

The Big-O is also $O(n^2)$.

Problem 3:

What is the big-Oh time complexity for searching an entire graph using *depth-first search* (DFS)? Does the representation of the graph make a difference? Justify your answer.

.

ANS

Assume that the number of vertices of a graph is ***n**,* and there are ***e*** edges in it.

When traversing the graph using DFS, for each node we only invoke the DFS function once. It is because once a node has already been visited and marked, we won't search from it again. For each node *x,* we consider all other nodes *y* which *x* has directed-edges in to.

If the representation of the graph is *an adjacency list,* the big-Oh time complexity for searching an entire graph is $O(n+e)$

If the representation of the graph is *an adjacency matrix,* the big-Oh time complexity for searching an entire graph is $O(n^2)$

Problem 4:

What is the big-Oh time complexity for searching an entire graph using *breadth-first search* (BFS)? Does the representation of the graph make a difference? Justify your answer.

ANS

Assume that the number of vertices of a graph is ***n**,* and there are ***e*** edges in it.

BFS is a procedure that borrows queues for storage, looking up hierarchically. When traversing the graph using BFS, we will only push each node in the queue for once. For each node *x,* we consider all other nodes *y* which *x* has directed-edges in to, once a node has already been visited and marked, we won't search from it again.

The BFS have a different order for visiting nodes with DFS, it will access to points close to the starting point first. So the big-Oh time complexity is same as DFS.

If the representation of the graph is *an adjacency list,* the big-Oh time complexity for searching an entire graph is $O(n+e)$

If the representation of the graph is *an adjacency matrix,* the big-Oh time complexity for searching an entire graph is $O(n^2)$

Homework 9 - shortest path in graph

Problem 1:

What is the big-Oh space complexity of Dijkstra's? Justify your answer.

ANS

Based on the implementation in the homework, the representation of the graph is an adjacency list, and using the a circular queue to implement Dijkstra’s algorithm.

Assume that the number of vertices of a graph is ***n****,* and there are ***e*** edges in it. The big-Oh space complexity is $O(n + e)$.

We define one struct graph_node_t for each node, another struct graph_edge_t for each edge in the graph. All ***n*** nodes of the graph are stored in a Double Linked List, for each node there are two Double Linked Lists storing the out neighbors $\boldsymbol{e_{to}}$ **(<*e*)** edges and in neighbors $\boldsymbol{e_{from}}$**(<*e*)** edges.

For the implementation of the algorithm, we use one Double Linked List to store visited nodes, meanwhile using one Circular Queue to store un-visited nodes. The maximum size of DLL or Queue is ***n.***

In the worst case, the number nodes and edges stored in DLLs or Queue would be equal to ***n*** and ***e.*** Thus, the largest number of nodes is ***an,*** *which is* ⇒ $\boldsymbol{n}$**,** the largest number of edges is ***be,*** *which is* ⇒ ***e.*** The big-Oh space complexity is $O(n + e)$.

Problem 2:

What is the big-Oh time complexity for Dijkstra's? Justify your answer.

ANS

Based on the implementation in the homework, the representation of the graph is an adjacency list, and using the a circular queue to implement Dijkstra's algorithm.

Assume that the number of vertices of a graph is ***n****,* and there are ***e*** edges in it. The big-Oh time complexity is $O(n^2)$.

For each u in Graph: u.cost =INFINITY run **n** times.
Pushing all nodes to Q, all cost is infinity runs **n** times.

For the implementation of the Dijkstra's algorithm, we create queue for un-visited nodes storage, the size is the number of all nodes in the graph ***n.***
The first loop that while *queue is not empty* need to operate ***n*** times, at each time we will pop one node with minimum cost and push it into visited double linked list.
The second loop while $position < queue -> size, position++ \; size --$, the operation time is like $n + n(n-1) + (n-2) \ldots . + 1 = \frac{n}{2}$ times. The third loop while traverse all nodes in visited dll, the operation time is like $1 + 2 + 3 \ldots + (n-1) + n = \frac{n}{2}$ times. For both second and third, the largest operation times is also ***n.***

Thus, the time of the implementation for Dijkstra's algorithm is $\boldsymbol{n^2}$

The total time would be $O(kn^2 + mn + ce)$. The big-Oh time complexity is $O(n^2)$.

Problem 3:

Write up the *proof by induction* of the correctness of Dijkstra's algorithm. .

ANS

Assume that the number of vertices of a graph is ***n****,* and there are ***e*** edges in it. It is a directed graph with non-negative edge weights,
One double liked list is created to store nodes which might be reachable with the minimum cost for the start node. One circular queue, with size ***n***, is the storage for un-visited nodes of graph.

Initialization
For Graph[G] vertex **n**,edge **e**
all graph $\text{nodes} -> \text{cost} = \infty$, put all nodes into queue Q={start, a, b,...x}
$\text{start} -> \text{cost} = 0$, dll visited L={start},

There will be three types of nodes in list L
* From start node to itself, the cost is zero
* the node has a shortest path with start node, the cost is min[start.cost+edge.cost]
* the node is not reachable to start node, the cost is infinity.

Proof by induction

#begin proof

Proposition: we prove this fact by induction that the Mth node that's indexed into visited list L, all *List_i i*n L must have found the shortest path *Short[List_i]* and it is equals to *Cost[List_i]*

While M = 1, List={star} =>Cost[start] ==short[start]==0；

Let U be the chosen node indexed into L next.

While M=2, there must be a edge contained directly between start and U pushing into the list L. The shortest path is the edge from the start node to first node, Cost[u]=start.cost+ cost(start,u), which is definitely smaller than infinity.

Then Assume that while M = k, all nodes in list L have found the shortest path, the fact is true.

Then let V be the chosen node indexed into L next.

While M = k+1, find the smallest cost in Q, return the corresponding nodes V. V which is the node going through other nodes List_i has the short distance reached by start. This path A could be like start -> U ->V, we need to prove Cost[v]==short[v].

#contradiction begin

Assume that the fact is false, which means there exists another path with shorter distance from start to V, this path B could be like $\text{start} -> X -> Y ->$V, this path is the short[v].

This path go through the Y which is un-visited node in Queue, the Y could be Y or some others un-visited nodes. For now, distance of path $B == \text{Cost}[y] + \text{Cost}(y, v) == \text{short}[v]$.

Cause Y is un-visited, $\text{Cost}[v] <= \text{Cost}[y]$

$$=> \text{Cost}[v] < B == \text{short}[v]$$

Cost[V] is the shortest path comparing with B instead of Cost[y]+Cost(y,v).

Thus, the assumption is invalid.There is not another path with shorted distance from start to V.

#contradiction end

Thus, while M=k+1 V which is the node going through other nodes List_i has the short distance reached by start. This path A could be like $\text{start} -> U ->$V, Cost[v]==short[v].

The proposition is true.

#proof end.

Lab 5 - Proofs

In this lab, we are going to work to become comfortable with the CLRS notation and answering basic problems about algorithmic performance and correctness

Problem 1: Insertion Sort, Descending order

Using CLRS's notation, rewrite pseudocode for Insertion Sort so that it sorts in descending order:

There is an array A[1....n]

The number *n* of elements in *A* is denoted by *A.length.*

```
INSERTION-SORT(A)
for j = 2 to A.length
    Key = A[j]
    // Insert A[j] into the sorted reverse A[j-1...1]
    i = j - i
    While i > 0 and A[i] <key
        A[i+1] =A[i]
        i = i - 1
    A[i + 1] = key
```

Problem 2: Linear Search pseudocode

We're going to define the searching problem in CLRS terms.

Input: A sequence of *n* numbers A = ⟨ $a_1, a_2, \ldots a_n$ ⟩

Output: An index *i* such that *v = A[i]* or the special value NIL if *v* does not appear in *A*.

Write pseudocode in CLRS style for linear search, which scans through the sequence, looking for *v*. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

Linear search(A)

Pseudocode

```
for i = 1 to A.length
        If A[i] == v
        return i
return NIL
```

A loop invariant

1) Invariant: we have traversed *i-1* elements of array *A.The subarray A =[1...i -1]* does not contain the target value *v.*

2) Initialization: *Before the first iteration of the loop, i =1 and t*he subarray *A* =[*1...i -1*] is empty.

3) Maintenance: For each iteration of the loop, we check *A*[*i*] is not the target number, showing that each iteration maintains the loop invariant.

4) Termination: If we terminate the loop with returning index *i*, it is when we already find an element *A*[i] equals to the value *v.* In this case, our algorithm is correct.

If we terminate the loop without returning index *i*, which means there is not an element of *A*[*1...n*] equals to the value *v.* And we return NIL as required. Hence, the algorithm is correct.

Problem 3: Selection Sort

In CLRS notation, write pseudo for selection sort. Assume that selection sort works this way: It sorts *n* numbers stored in array *A* by first finding the smallest element of *A* and exchanging it with the first element in *A*. Then it finds the second smallest item in *A* and exchanges it with the second item in *A*. It continues for the first *n-1* elements in *A*.

Write pseudocode for selection sort.

SELECTION-SORT(A)

```
Function Swap(a, b)
        temp = a
        a = b
        b = temp
for i = 1 to A.length - 1
        key = i
        for j = i + 1 to A.length
            If A [j] < A [key]
                key = j    //record the index of minimum
        Swap A[key] and A[i]
```

What loop invariant does this algorithm maintain?

ANS Invariant: At the start of each iteration of the for loop , the *A*[*1...i -1*] we have traversed has *i-1* elements in sequence.

Why does it need to run for only the first *n-1* elements, instead of for all *n* elements?

ANS At the first iteration, we take the first item *i* to be the smallest one, and then comparing it with the rest element of the array. After *n-1* elements have been sorted. The last one element is the biggest one, so we do not need to run for all *n* elements.

Bonus: Problem 4: Why we never consider best case

How can we modify any sorting algorithm to have a good best case running time, returning a definitely sorted array in O(n)? Remember that, in the best case, you can assume the data is in the most beneficial format that you need for your solution to work.

ANS

We can define a *flag* in for loop of the sort algorithm, and initialize the value of *flag equals to 0(which means true).* Also under the swap function line, we evaluate the *flag equals to 1(which means false)*.

If we have a sequence array, the swap function will never be operated, and the flag will still equals 0(true).

Add the statement *if flag = 0,* break the for loop. The big-Oh complexity of best case is *O(n).*

lab04-exercises

Author’s name: Cuiting Huang

Partner: Hongyang Ye, Siqi Chen

Author affiliation: Northeastern University

City & country location: Zhengzhou, China

E-mail address: huang.cui@northeastern.edu

Abstract – This is the written exercises of lab04 sorting

Index Terms – Sorting, Bubble, Selection, Big O

Question1

Explain what you think the worst-case, big-Oh complexity and the best-case, big-Oh complexity of bubble sort is. Why do you think that?

ANS

Worst-case: We have a reversed array that every single element needs to be sorted. And the big-Oh complexity of this case is O(n^2).

For the worst-case we need to set every step costs a constant time, we need n steps to sort a single element from the beginning to the end. For n elements in an array, in total we need n*n steps.

Best-case: We have a sequential array that all the elements are already sorted. And the big-Oh complexity of this case is O(n).

For every elements have already been sorted, we can just walk through all the elements for a time.

i.e., we run the internal for loop for one time, which takes n steps. If we found all the elements are sorted, we break the loop.

Question2

Explain what you think the worst-case, big-Oh complexity and the best-case, big-Oh complexity of selection sort is. Why do you think that?

ANS

For selection sort, we could only know which element is the smallest after the iteration of the end of the unsorted portion. Even if the array is sequential sorted ,we must also iterate until the end.

For one element, we conduct function find Minimum for one time, which takes n steps.

For every elements, it needs n*n steps.

There is no difference between every case of the selection sort. The big-Oh complexity of selection sort is always O(n^2).

Question3

Does selection sort require any additional storage (i.e. did you have to allocate any extra memory to perform the sort?) beyond the original array?

ANS

No. According to our code, we do not allocate any extra memory to perform the sort.

Question4

Would the big-Oh complexity of any of these algorithms change if we used a linked list instead of an array?

ANS

The big-Oh complexity remains the same for a linked list.
For bubble sort, since it exchanges the locations of two adjacent elements. It works the same for two adjacent nodes in a linked list.
For selection sort, we can use two pointers to change the location of the selected element and the minimum. Therefore, the big-Oh complexity also remains the same.

Question5

Explain what you think big-Oh complexity of sorting algorithm that is built into the c libraries is. Why do you think that?

ANS

I think the big-Oh complexity of sorting algorithm that is built into the c libraries is O(nlogn).
According to the runtime of these sorts we got in the google sheet. The csort always runs so faster than the others. For the best case of bubble sort,the big-Oh complexity of this case is O(n). I guess the sort built in C-library should be more quicker, and O(nlogn) is a good choice.

Lab 6 - Merge Sort

Problem 1:

Explain what you think the worst-case, big-Oh complexity and the best-case, big-Oh complexity of merge sort is. Why do you think that?

ANS

The worst case can be that each element in an array needs to be sorted, and the best case is that the array is already sorted. Both worst-case and best-case, it has the same big-Oh complexity $O(nlogn)$.

The complexity of merge sort equals to subarray sorting time plus merge time. According to the pseudo code of merge sort, we can see that whether the element in an array is located at the right place, and the array is already sorted or not, we always halve the array into shorter pieces recursively, until there is only one element in a subarray, and then combine all the subarrays.

Therefore, the worst case and best case as a matter of fact do not affect the big-Oh complexity of merge sort.

Assuming that dividing the array and combing the array take in total cn time, and the total time quired by merge sort is $T(n)$. For each division, we are actually halving the array, and for each smaller array, it needs $\frac{1}{2}T(\frac{n}{2})$ time.Then we have

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

$$= 2(2T\left(\frac{n}{4}\right) + \frac{cn}{2}) + cn$$

$$= 2\left[2\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + \frac{cn}{2}\right] + cn$$

$$\ldots \quad = 2^k T\left(\frac{n}{2^k}\right) + kcn$$

Let $n = 2^k, k = \ log_2 n$

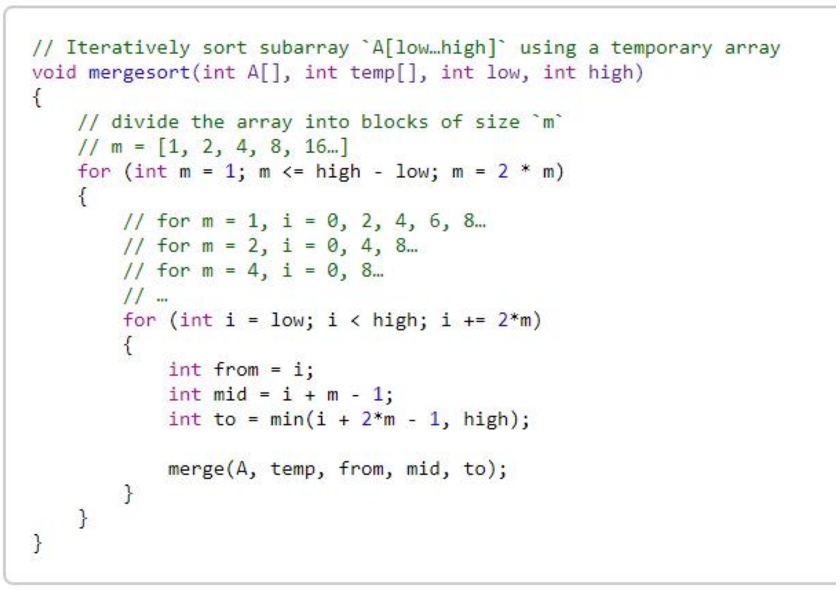
$$T(n) = nT(1) + cnlog_2 n = n + cnlog_2 n$$

$$T(n) \in O(nlogn)$$

Therefore, we conclude that both the worst-case and best-case big-Oh complexity of merge sort are $O(nlogn)$.

Problem 2:

Merge sort as we have implemented in there is a recursive algorithm as this is the easiest way to think about it. It is also possible to implement merge sort iteratively:



```
// Iteratively sort subarray `A[low…high]` using a temporary array
void mergesort(int A[], int temp[], int low, int high)
{
    // divide the array into blocks of size `m`
    // m = [1, 2, 4, 8, 16…]
    for (int m = 1; m <= high - low; m = 2 * m)
    {
        // for m = 1, i = 0, 2, 4, 6, 8…
        // for m = 2, i = 0, 4, 8…
        // for m = 4, i = 0, 8…
        // …
        for (int i = low; i < high; i += 2*m)
        {
            int from = i;
            int mid = i + m - 1;
            int to = min(i + 2*m - 1, high);

            merge(A, temp, from, mid, to);
        }
    }
}
```

Explain what you think the worst-case, big-Oh complexity and the best-case, big-Oh complexity is for this iterative merge sort. Why do you think that?

ANS

Both the worst-case and best-case big-Oh complexity of iterative merge sort are $O(nlogn)$.

The analysis of iterative merge sort is quite the same as that of recursive merge sort. From the codes above, we can see that in the iterative merge sort, the array is also divided into small pieces. Then all the subarrays are combined together to obtain a sorted array.

The only difference between recursive merge sort and iterative merge sort is that the recursive merge sort starts from halving the array. While the iterative merge sort starts from the smallest pieces of the array, that is subarray only containing one element.

No matter we are applying recursion or iteration in a merge sort, the underlying logic is always divide and conquer. And for iterative merge sort, the best case and the worst case also make no difference because even if the array is already sorted, we still have to starting from the smallest pieces and combining them once and once again.

Therefore, we conclude that both the worst-case and best-case big-Oh complexity of iterative merge sort are $O(nlogn)$.

Lab 7 - DFS and Trees

Problem 1:

What data type in the C programming language allows for the largest values of factorial to be computed?

ANS

According to the reading material given in the lab,https://www.geeksforgeeks.org/data-types-in-c/

The data type allowing for the largest values of factorial to be computed is the *unsigned long long.* The big number of its range is 18 446 744 073 709 551 615.

Problem 2:

At what input value for the recursive factorial function does your computer start to 'crash' or really slow down when you try to compute a factorial? Is it the same value as the iterative function? Experiment and report your result.

ANS

As for the recursive factorial function, when the argument became 1000000, it crashed. It is not the same value as the iterative function.

As for the iterative factorial function, it did not crash when the argument became $1 * 10^{46}$, though we got the warning.

Problem 3:

In 2-3 sentences, describe why you believe you saw or didn't see differences between the iterative and recursive versions of factorial.

ANS

The recursive can be implementation while it could be easily done in a simple loop, it can be able to write the same code in 5 lines instead of 20 is a huge deal.

Recursive functions have to keep the function records in memory and jump from one memory address to another to be invoked to pass parameters and return values. That makes them very bad performance wise than iterative.

Lab08 - Matrix

Problem 1:

We have focused primarily on *time complexity* in this course, but when choosing data structures, space complexity is often as important of a constraint. Given an adjacency matrix, what is the 'space complexity' in Big-O. That is, given *n* nodes, how much space (i.e. memory) would I need to represent all of the relationships given. Explain your response.

ANS

For each vertex, it takes *n* blocks of space, and its adjacency matrix takes n^2 blocks of space to represent all of the relationships given.

Therefore, given n nodes, the total space required by the adjacency matrix is $n^2 + n$.

The space complexity in Big-O is n^2.

Notice that undirected adjacency matrix is symmetrical, one of its diagonals from left top to right bottom is always zero, so it takes $\frac{n(n-1)}{2}$ blocks of space to represent all of the relationships given. The Big-O is also n^2.

Problem 2:

Will it ever make sense for ROWS != COLUMNS in an adjacency matrix? That is, if we want to be able to model relationships between every node in a graph, must rows always equal the number of columns in an adjacency matrix? Explain why or why not.

ANS

it doesn't make sense if ROWS != COLUMNS in an adjacency matrix.

An adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph[1]*.*

All the relationships between the nodes is n^2, so ROWS must equals to COLUMNS.

[1] Adjacency matrix https://en.wikipedia.org/wiki/Adjacency_matrix

Problem 3:

Can you run topological sort on a graph that is undirected?

ANS

We can't run topological sort on a graph that is undirected.

Topological sort on a graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, u comes before v in the ordering.

There is at least one loop in undirected graph, the relationship between the nodes is not linear. When we try to run tsort with such graph, It is fail to output all relationships in graph.

Problem 4:

Can you run topological sort on a directed graph that has cycle?

ANS

We can't run topological sort on a directed graph that has cycle.

It has the same reason as #3. Because there is at least loop in graph, the relationship between the data is inconsistent.

Problem 5:

For question number 4, how would you know you have a cycle in a graph? What algorithm or strategy could you use to detect the cycles? **Hint** we have already learned about this traversal.

ANS

We could use DFS algorithm to detect whether there is a cycle in the graph.

*Here's **an excerpt** from wikipedia's pseudocode of Topological sorting:*

```
L ← Empty list that will contain the sorted nodes
while exists nodes without a permanent mark do
    select an unmarked node n
    visit(n)

function visit(node n)
    if  n has a permanent mark then
        return
    if n has a temporary mark then
        stop    (not a DAG)

    mark n with a temporary mark

    for each node m with an edge from n to m do
        visit(m)

    remove temporary mark from n
    mark n with a permanent mark
    add n to head of L[2]
```

For each node *n,* we consider all other nodes *m* which *n* has directed-edges in to. Then we add *n* to list *L.* If the graph has a cycle which means there is a node *m* has directed-edges in to *n.* Since each edge and node can only be visited once. We can definitely tell if the graph has a cycle or not.

[2] Topological sorting https://en.wikipedia.org/wiki/Topological_sorting