Data Structures and Algorithms
Week 2 problem sheet

## A. Linked List data structures

1. Give three points of differences between an array and a linked list.
*
* **Answer:**
	+ The size of an array is fixed when it is constructed, but a linked list can grow and shrink.
	+ An array provides very fast ($O\left(n\right)$) access to any element in the array; however, nodes in a linked list can only be accessed by following links to them from the head of the list.
	+ Unused elements in an array still occupy memory; however, items removed from a linked list will be garbage-collected by the Java Virtual Machine.
	+
1. Draw the **stack** data structure, when a linked list implementation is used, for each step in the following sequence:
* push(1), push(2), pop, push(3), push(4),
pop, pop, push(5).
*
*
* **Answer:**
* 
* 
* 
* 
* 
* 
* 
* 
*
1. Draw the **queue** data structure, when a linked list implementation is used, for each step in the following sequence:
* enqueue(1), enqueue(2), dequeue, enqueue(3), enqueue(4),
dequeue, dequeue, enqueue(5).
*
*
* **Answer:**
* 
* 
* 
* 
* 
* 
* 
* 
*
1. A cyclic linked list implementation of a Queue:
* An (unbounded) queue can be implemented cyclically based on a linked representation. In this case, rather than referencing “null”, the successor of the last item in the queue references the beginning of the queue. While this is not necessary to prevent memory erosion, it does mean that rather than having two references to the beginning and the end of the queue, only a *single* reference is needed.
* Write a cyclic linked implementation of a queue called QueueLinked using this approach.
* Your queue ADT must implement the QueueADT interface.
* Fully document your code.
1. A double-ended queue (*deque*) of characters differs from a standard queue in that it allows objects to be added and deleted from both ends of the queue. Contrast this to a standard queue, where objects can only be added to the end of the queue and removed from the front.
* Write a singly linked-list implementation of the deque ADT in Java.
* Your implementation should contain the following methods:
	+ DequeCharCyclic(s): create an empty deque of size s.
	+ isEmpty(): return true iff the deque is empty, false otherwise.
	+ isFull(): return true iff the deque is full, false otherwise.
	+ pushLeft(c): add character c as the left-most character in the deque, or throw an Overflow exception if the deque is full.
	+ pushRight(c): add character c as the right-most character in the deque, or throw an Overflow exception if the deque is full.
	+ peekLeft(): return the left-most character in the deque, or throw an Underflow exception if the deque is empty.
	+ peekRight(): return the right-most character in the deque, or throw an Underflow exception if the deque is empty.
	+ popLeft(): remove and return the left-most character in the deque, or throw an Underflow exception if the deque is empty.
	+ popRight(): remove and return the right-most character in the deque, or throw an Underflow exception if the deque is empty.
1. **Challenge:** See extra questions on dequeues at [**http://teaching.csse.uwa.edu.au/units/CITS2200/Tutorials/tutorial04.html**](http://teaching.csse.uwa.edu.au/units/CITS2200/Tutorials/tutorial04.html).

## B. Big “O” notation

1. Assume that each of the following expressions each represent the worst-case time taken by some algorithm, in terms of the size of the input, $x$.
* Group the expressions into equivalent levels of Big-“O” complexity.
* $x^{2}$, $x$, $x^{2}+x$, $x^{2}−x$, and $\frac{x^{3}}{x−1}$.
* **Answer:**
* The expression $x$ has $O\left(n\right)$ (linear) complexity.
* All the others have $O\left(n^{2}\right)$ (quadratic) complexity.
1. Solving a problem requires running an O(N) algorithm and then afterwards a second O(N) algorithm. What is the total cost of solving the problem? Why?
* **Answer:**
* It has $O\left(N\right)$ complexity, since constant multipliers are ignored in Big “O” notation; therefore an algorithm with complexity $N+N=2N$ has complexity of order $O\left(N\right)$.
1. Solving a problem requires running an O(N) algorithm and then afterwards an O(N2) algorithm. What is the total cost of solving the problem? Why?
* **Answer:**
* It has $O\left(N^{2}\right)$ complexity, since as $N$ increases towards infinity, the $N^{2}$ term will predominate; therefore an algorithm with complexity $N^{2}+N$ has complexity of order $O\left(N^{2}\right)$.
1. In terms of n, what is the running time of the following algorithm to compute x to the power n (xn)? Can you think of a faster approach?
* /\* calculates x to the n \*/
 public static double power( double x, int n ) {
 double result = 1.0;

 for( int i = 0; i < n; i++ ) {
 result = result \* x;
 }
 return result;
 }
* **Answer:**
* It has $O\left(N\right)$ complexity, since the loop is executed n times.
* A faster approach is to rely on the fact that $x^{2m}=x^{m}×x^{m}$:
* /\* calculates x to the n \*/
 public static double power( double x, int n ) {
 double result = x;
 int i = n;
 while (i > 1) {
 result = result \* result;
 i = i / 2;
 }
 if (i % 2 == 1)
 result \*= x;
 return result;
 }
* Each time round the loop, the power that we’ve calculated in result doubles; therefore the complexity of this solution will be $O\left(logn\right)$.
1. Which of the following statements make sense or not? Why?
	1. My algorithm has complexity $O\left(2N^{2}\right)$
	2. My algorithm has complexity $O\left(N^{2}+N\right)$
	*
* **Answer:**
	1. This makes sense, but would be better written as $O\left(N^{2}\right)$, since Big “O” complexity ignores constant multipliers.
	+
	1. This makes sense, but would be better written as $O\left(N^{2}\right)$, since as $N$ increases towards infinity, the $N^{2}$ term will predominate and the $N$ term can be ignored.
	+
* See Weiss Ch 6 “Common Errors” (at the end of the chapter) for more explanation of these points.
1. Are the following statements true or false? Why?
	1. A method with one loop nested inside another must have complexity $O\left(N^{2}\right)$
	2. If method A has complexity $O\left(N\right)$ and method B has complexity $O\left(N\right)$ then an algorithm which performs A followed by B is also $O\left(N\right)$
*
* **Answer:**
	1. This is false. One issue is that there might be *more* loops, and the complexity could be greater than $O\left(N^{2}\right)$. Although in many cases, a method with one loop nested inside another will have $O\left(N^{2}\right)$ complexity, there are sitations in which it will not.
	+ Even if we assume the code in the method does not make calls to another method (which could have, say, $O\left(n\right)$ complexity, thus increasing the complexity of the calling method), it still might not have $O\left(n^{2}\right)$ complexity. For instance, the inner loop might have an empty body, or never execute. Consider the following code, where the body of the inner loop never executes:
	+ for(int i = 0; i < n; i++) {
	 for(j = 99; j < 1; j++) {
	 // some code
	 }
	 }
	1. This is true. The time taken by the algorithm will be $2N$, which means the complexity of the algorithm is $O\left(N\right)$.
* See Weiss Ch 6 “Common Errors” (at the end of the chapter) for more explanation of these points.
1. Consider an **array implementation** of the stack ADT. Give a short description of an implementation for each of its functions in words. What is the Big “O” complexity of each of these operations, and why?
	* isEmpty
	* isFull
	* pop
	* push
*
* **Answer:**
	+ isEmpty: this checks whether the number of elements is 0
	+ isFull: this checks whether number of elements is same size as the array
	+ pop: this returns the element on the top of the stack, and updates the top-of-stack index.
	+ push: this adds an element to the top of the stack, and updates the top of stack index.
	+
* All these operations are O(1) because they take constant time. They do not depend on the size of the stack.
1. The following method searches an array (stored in “block”) to see whether any item appears twice. If so, it returns true. If no duplicates are found it returns false.
* public boolean hasMatch (int[] block) {
 boolean found = false;
 for (int i=0; i < block.length; i++) {
 for (int j=0; j < block.length; j++) {
 found = found ||
 (i != j && block[i]==block[j]);
 }
 }
 return found;
 }
* If the function $f\left(n\right)$ describes the time performance of the hasMatch method, where $n$ denotes the size of the parameter block, which of the following is the smallest possible Big “O” for $f\left(n\right)$? Why?
	1. f(n) is $O\left(1\right)$
	2. f(n) is $O\left(logn\right)$
	3. f(n) is $O\left(n\right)$
	4. f(n) is $O\left(n^{2}\right)$
*
* **Answer:**
* The correct answer is (d).
* The outer loop is executed $n$ times, and for each of those $n$ times, the inner loop is executed $n$ times. So the complexity for the method is $O\left(n^{2}\right)$.
1. Write the simplest algorithm you can think of to determine whether an integer i exists such that $A\_{i}$ = i in an array, A, of increasing integers.
* Now, try to give a more efficient algorithm, explaining your reasoning. What is the Big “O” running time for each of your algorithms?
* **Answer:**
* Most people would regard the *simplest* solution as being sequential search.
* // return the index of an element were A[i] == i, else -1
 public static int searchArray(int[] A) {
 for (int i=0; i<A.length; i++) {
 if (A[i]==i)
 return i;
 }
 return NOT\_FOUND; // NOT\_FOUND = -1
 }
* This will have complexity $O\left(n\right)$.
* For a faster solution, we reason about the problem as follows.
* We are looking for an element at index $i$ where A[i] == i. Suppose we look at some array element and see that A[i] > i. We know that the elements are strictly increasing; therefore the element we’re after *can’t* be in any of the subsequent elements, since for those elements, too, it will be the case that A[i] > i.
* Similar reasoning applies if we look at an array element and see that A[i] < i; we then know that the element we’re after can’t appear at any of the indices *lower* than $i$.
* Consequently, we can use an algorithm similar to binary search to find the element we’re after. This will have complexity $O\left(logn\right)$, since at each step, we halve the portion of the array in which the item we want could appear.
* A possible solution is therefore:
* // return the index of an element were A[i] == i, else -1
 public static int searchArray(int[] A) {
 int low = 0;
 int high = A.length - 1;
 int mid;
 while (low <= high) {
 mid = (low + high) / 2;
 if (A[mid] == mid) { // we've found it
 return mid;
 }
 // otherwise keep looking
 if (A[mid] < mid) {
 //not in lower so now search only upper
 low = mid + 1;
 } else if (A[mid] > mid) {
 //not in upper so now search only lower
 high = mid - 1;
 }
 }
 // if we get here, the item was not found
 return NOT\_FOUND; // NOT\_FOUND = -1
 }
1. The method methodX searches an array as follows.
* public boolean methodX (int[] block) {
 boolean found = false;
 for (int i=0; i<block.length; i++) {
 for (int j=0; j<block.length; j++) {
 found = found || block[i]==block[j];
 }
 }
 return found;
 }
* Which of the following is true of this function? Why?
	1. It never returns true.
	2. It returns true only if the same item appears twice.
	3. It returns true if the last two items compared are the same.
	4. It always returns true.
*
* **Answer:**
* The correct answer is (b).
1. Method hasTwoTrueValues returns true if at least two values in an array of Booleans are true. What is the Big “O” running time for all three implementations proposed below?
* **Version 1:**
* public boolean hasTwoTrueValues(boolean[] arr) {
 int count = 0;
 for(int i = 0; i < arr.length; i++)
 if (arr[i])
 count++;
 return count >= 2;
 }
* **Version 2:**
* public boolean hasTwoTrueValues(boolean[] arr) {
 for(int i = 0; i < arr.length; i++)
 for(int j = i + 1; j < arr.length; j++)
 if( arr[i] && arr[j] )
 return true;
 return false;
 }
* **Version 3:**
* public boolean hasTwoTrueValues(boolean[] arr) {
 for(int i = 0; i < arr.length; i++) {
 if( arr[i] )
 for(int j = i + 1; j < arr.length; j++)
 if( arr[j] )
 return true;
 }
 return false;
 }
* **Answer:**
* ***Version 1*** contains a single loop, which iterates over all the elements of the array. Therefore version 1 will have complexity $O\left(N\right)$, where $N$ is the length of the array.
* ***Version 2*** contains a nested loop – so our initial guess might be that its complexity is $O\left(N^{2}\right)$, where $N$ is the length of the array arr. We can check our reasoning by considering what will happen if we search an array of, say, length 7, when the array does not contain any true values.
* The variable i will take on the values 0, 1, 2, 3, 4, 5 and 6 in order. How many times will the inner loop execute? When i is 0, it will execute 6 times (j will take on values from 1 to 6); when i is 1, it will execute 5 times (j will take on values from 1 to 5); and so on, until when i is 6, the inner loop will not execute at all.
* So the number of times the body of the inner loop will execute is 6 + 5 + 4 + 3 + 2 + 1 + 0.
* We can generalize this, and deduce that for an array of size $N$, the body of the inner loop will execute $0+1+...+\left(N−1\right)+N$ times.
* We know that there is a simple formula for
* $0+1+...+\left(N−1\right)+N$
* – it was covered in the lecture slides on Insertion sort (see the slides labelled “Complexity of insertion sort”). The formula is
* $\frac{N×\left(N−1\right)}{2}$
* which is equivalent to
* $\frac{N^{2}−N}{2}$
* Therefore, we can conclude that the complexity of this algorithm is $O\left(N^{2}\right)$.
* ***Version 3*** is very similar to version 2, except that the inner loop is only executed when arr[i] is true in the outer loop. The *worst*-case performance of version 3 is exactly the same (it is $O\left(N^{2}\right)$), because if the array does not contain any true values, the body of the inner loop will be executed exactly the same number of times as for version 2.
* However, the *best*-case performance of version 3 will be better than for version 2, because version 3 avoids running the inner loop except where necessary.

## D. Recursion



1. Write a recursive method that calculates factorial of a positive number. Choose a suitable exception for its error cases.
* **Answer:**
* // return factorial of a positive number n
public static long factorial(int n) {
 if (n < 0)
 throw new IllegalArgumentException("argument can't be negative");
 if (n == 0)
 return 1;
 return n \* factorial(n-1);
}
1. The $n$th harmonic number is the sum of the reciprocals of the first $n$ positive natural numbers. So $H\_{n}=1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+...+\frac{1}{n}$
* For example,
* 
* Explain what is *wrong* with each of the following three definitions of a recursive method to calculate the $n$th harmonic number. Then write a correct Java implementation and test it.
* **Version 1**
* public static double H(int N) {
 return H(N-1) + 1.0/N;
 }
* **Version 2**
* public static double H(int N) {
 if (N == 1) return 1.0;
 return H(N) + 1.0/N;
 }
* **Version 3**
* public static double H(int N) {
 if (N == 0) return 0.0;
 return H(N-1) + 1.0/N;
 }
* **Answer:**
* The problem with ***version 1*** is that it doesn’t have a base case – regardless of what value $N$ is passed in, H will *always* be called again, resulting in an infinite loop. (Or in practice, a stack overflow exception on the Java Virtual Machine.)
* The problem with ***version 2*** is that it doesn’t progress towards a base case; if called with argument N, it will make a recursive call to H with exactly the same argument N – so the recursion will never finish.
* The problem with ***version 3*** is that it simply doesn’t correctly implement the formula given. Harmonic numbers are only defined for positive natural numbers, not for 0, so it is incorrect to write code which returns a result for an argument of 0.
* A problem with *all* of the implementations is that they do no error-checking to make sure that they have been called with a positive number.
* A correct implementation would be as follows:
* // return nth harmonic number
public static double H(int N) {
 if (N < 1)
 throw new IllegalArgumentException("argument must be positive");
 if (N == 1)
 return 1.0;
 return H(N-1) + 1.0/N;
}
*
1. Write a recursive method that returns the number of 1s in the binary representation of N. Use the fact that this number equals the number of 1s in the representation of N/ 2, plus 1, if N is odd.
* First: what is the base case ? what is the step case?
* Second: express this recursion in a Java method.
* Third: write some test cases to test your code.
* (source: Princeton intro to cs)
* **Answer:**
* Our *base case* will be that $n$ is 0; in that case, the binary representation of $n$ contains no 1s.
* Our *recursive part* will work out whether the value of $n$ we’re passed is odd or not. If it is odd, we add 1 to our count, if it is not odd, we don’t. We then make a recursive call, passing n / 2 as the argument.
* // return number of 1's in binary representation of n
// (or 0, for negative numbers)
public static int numberOfOnes(int n) {
 if (n < 1)
 return 0;
 return (n % 2) + numberOfOnes(n / 2);
}