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.
2. 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).

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).

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, .

* Group the expressions into equivalent levels of Big-“O” complexity.
* , , , , and .

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?
2. 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?
3. 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;  
   }

1. Which of the following statements make sense or not? Why?
   1. My algorithm has complexity
   2. My algorithm has complexity
2. Are the following statements true or false? Why?
   1. A method with one loop nested inside another must have complexity
   2. If method A has complexity and method B has complexity then an algorithm which performs A followed by B is also



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



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 describes the time performance of the hasMatch method, where denotes the size of the parameter block, which of the following is the smallest possible Big “O” for ? Why?
  1. f(n) is
  2. f(n) is
  3. f(n) is
  4. f(n) is

1. Write the simplest algorithm you can think of to determine whether an integer i exists such that = 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?

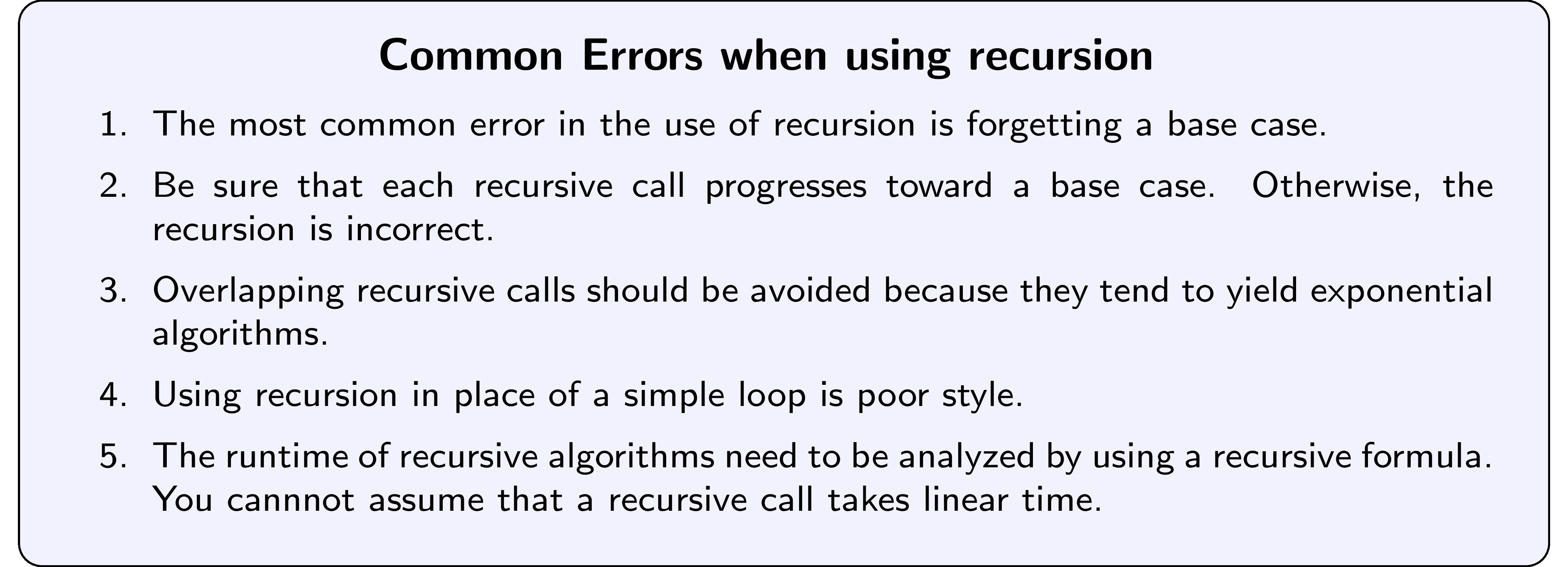
1. The method methodX searches an array as follows.

* 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.

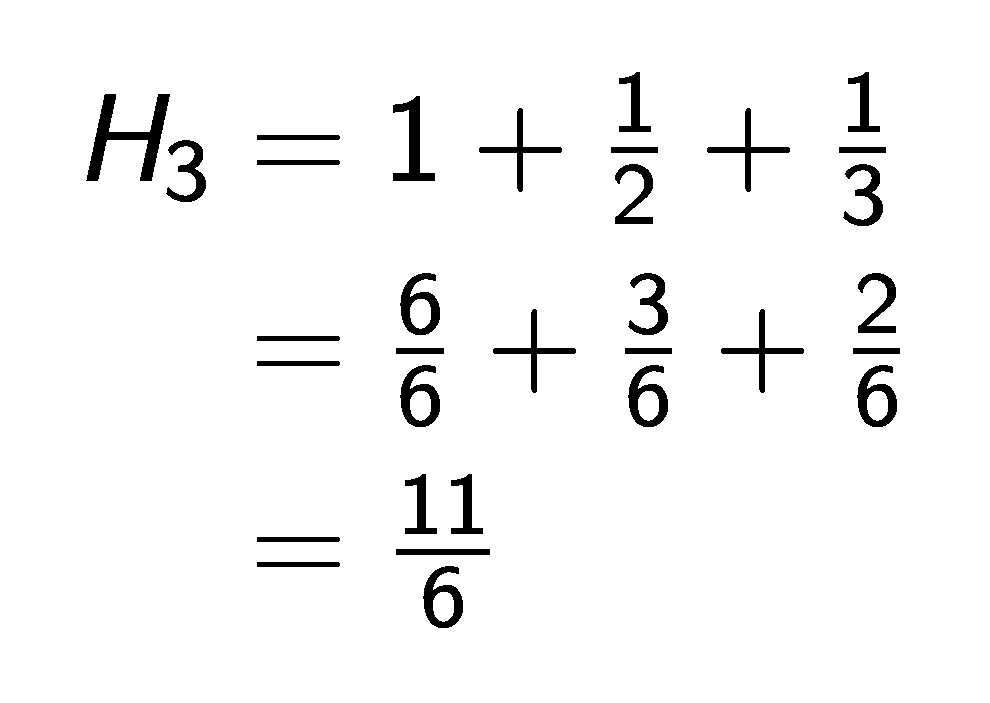
1. Method hasTwoTrueValues returns true if at least two values in an array of Booleans are true. What is the Big “O” running time for all three implementations proposed below?

* **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;  
   }

## D. Recursion



1. Write a recursive method that calculates factorial of a positive number. Choose a suitable exception for its error cases.
2. The th harmonic number is the sum of the reciprocals of the first positive natural numbers. So

* For example,
* 
* Explain what is *wrong* with each of the following three definitions of a recursive method to calculate the 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;  
   }

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)