ArrayLists:

ArrayList<String> names = new ArrayList<String>(); --use object classes(Integer, Double) for arraylists

Scanner input = new Scanner(new File(“words.txt”)); while (input.hasNext()) { String word = input.next(); allWords.add(word);

public static void mname (ArrayList<E> par) {...}

Iterator<Student> itr = students.iterator(); --iterate through an arraylist

while (itr.hasNext()) {

Student s = itr.next();

Abstract classes:

at least one abstract method, some design and some implementation

interfaces:

pure abstract classes

pure design

reading a file:

Scanner scan=**new** Scanner(**new** File("./common.txt"));

**while**(scan.hasNext()) {

common.add(scan.next());

}

making a file and print:

FileWriter fw = **new** FileWriter(input + "out");

PrintWriter pw = **new** PrintWriter(fw);

Recursion:

There are 2 key components to a recursive function:

1. the base case, and

2. the recursive decomposition

base case for which there’s a simple answer The base case is how the recursive function stops calling itself and begins to “unwind”

There must a recursive decomposition which breaks down the original problem into simpler problems of the same form The recursive decomposition must make progress towards the

base case If it doesn’t, then you have infinite recursion which will most likely result in a stack overflow!

Pseudo code: define recursiveFunction (parameters) {

if (baseCaseCondition) { // base case compute base solution without recursion }

else { // recursive decomposition decompose problem into smaller subproblems call recursiveFunction for each of the subproblems reassemble solutions into a solution for the whole } }

fact3: more efficient than linear recursion of same factorial idea

public static int fact3 (int n) {

return fact3Aux(n, 1);

}

private static int fact3Aux (int n, int result) {

if (n == 1) {

return result;

}

else {

return fact3Aux(n – 1, n \* result);

}

}

Algorithm analysis:

Let f(n) and g(n) be functions mapping non-negative

integers to real numbers. We say that “f(n) is in O(g(n))” if

there exists a real constant c > 0 and integer constant n0 ≥

1 such that:

f(n) ≤ cg(n), for all n ≥ n0

So, f(n)=O(g(n)) means f(n) is a member of the set O(g(n)).

Let f(n) and g(n) be functions mapping nonnegative integers

to real numbers. We say that “f(n) is in Ω(n)” if there is a

constant c > 0 and integer constant n0 ≥ 1 such that:

f(n) ≥ cg(n) for all n > n0

So, f(n)=Ω(g(n)) means f(n) is a member of the set Ω(g(n)).

Big-Omega describes the least amount of a resource that an

algorithm needs for some class of input.

Big-Omega denotes a lower bound.

“f(n) is in Θ(g(n))” if there are constants c1 > 0

and c2 > 0 and an integer constant n0 ≥ 1 such that:

0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0

In other words, the function f(n) belongs to the set of

Θ(g(n)) if there exist positive constants c1 and c2 such

that f(n) can be “sandwiched” between c1g(n) and c2g(n)

for sufficiently large n.

for Recursion: T(n)=T(n-1)+c or T(n)=T(n/2)+c etc.

merge sort: split and split array into individual integers or objects and then merge them into bigger and bigger arrays in order, recursive and base case is having the split finally into an array of one value

T(n)=2T(n/2)+cn T(n/2)=2T(n/4)+cn T(n)=4T(n/4)+2cn =8T(n/8)+3cn . . . =nT(n/n)+cn(log2n) =n+cn(log2n) O(nlog2n)

quick sort: partition and partition smaller and smaller sections of an array until the whole thing has been partitioned into the right order, recursive and base case is array of size 1. pick pivot to be boundary between upper and lower half of array. then recursively do the same stuff by calling the partition to both arrays

linked lists use pointers to objects which give the value we want