COMP 250 Crib Sheet
by Julian Lore Side 1 of 2

1 Algorithms

An algorithm is a **systematic** and **unambiguous** procedure producing an answer to a question/a solution in finite **number of steps**.
Algorithm has an input, sometimes, input needs to satisfy conditions. Algorithm has output (usually solution).

Good algorithms consist of correctness, speed, space & simplicity.
Correctness: right answer/right answer most of the time/close to right answer
Space: Amount of mem needed
Simplicity: easy to understand, analyze, implement, debug, modify, update

Iterative algorithms Problem solved by iterating (step-by-step), often using loops.

In-place algorithms Uses constant amount of memory (in addition of that used to store input). Important, because if data barely fits mem, don't want an algo using twice memory to sort.
Selection and Insertion in-place, just swapping.
MergeSort is not in-place, merge needs temporary array.
QuickSort can easily be made in-place

1.1 Binary Search

$O(\log n)$ Search for if something is in a list, like using a dictionary, split in half and check if lower or upper, then check corresponding half.
In array of n elements, a **key** k to search for **Out** array sorted in increasing order

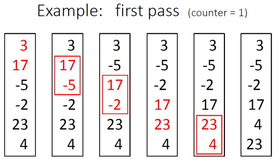
binarySearch(a,n,k)
left → 0
right → n
while $right > left + 1$ **do**
  mid ← $\lceil (left+right)/2 \rceil$
  if $A[mid] > k$ **then** right ← mid
  else left ← mid
if $A[left] = k$ **then** return True;
else return False;

1.2 Bubble Sort

$O(n^2)$
Sort # in ascending. Loop through list many times, if 2 elem next to each other wrong order, swap (need tmp var). ct is count, last N-2-ct elements already sorted on pass.

for $ct \leftarrow 1$ to $N-2$ **do**
  for $i \leftarrow 0$ to $N-2-ct$ **do**
    if $list[i] > list[i+1]$ **then**
      swap($list[i], list[i+1]$)

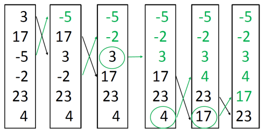
Example: first pass (counter = 1)



1.3 Selection Sort

$O(n^2)$
Sort # in ascending. Partition list into 2, sorted and remain. Find smallest from remain and add to sorted and so on. (SWAP)

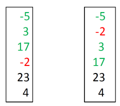
for $i \leftarrow 0$ to $N-2$ **do**
  $tmpIndex \leftarrow i$
  //i is first element in rest
  $tmpMinValue \leftarrow list[i]$
  for $k = i+1$ to $N-1$ **do**
    if $list[k] < tmpMinValue$ **then**
      $tmpIndex \leftarrow k$
      $tmpMinValue \leftarrow list[k]$
  if $tmpIndex != i$ **then**
    swap($list[i], list[tmpIndex]$)

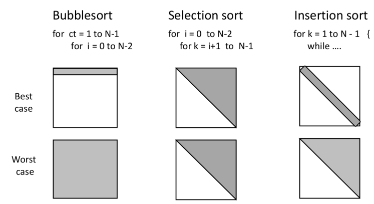


1.4 Insertion Sort

$O(n^2)$ for worst, $O(n)$ for best. Insert index k into correct position wrt 0 to k-1. i.e. 0 to k-1 already sorted, insert k at proper position

for $k \leftarrow 1$ to $N-1$ **do**
  $elementK \leftarrow list[k]$ // Store kth element, will overwrite
  $i \leftarrow k$
  while $i > 0$ and $list[i-1] > elementK$ **do**
  // $i > 0$ first to avoid out of bound
  //Shift everything bigger than kth to the right to fit k
    $list[i] \leftarrow list[i-1]$
    $i \leftarrow i-1$
  $list[i] \leftarrow elementK$





2 Multiplication Algorithms

Iterative , add b, a times.

Standard Grade school multiplication.
$a = a_0 a_1 \dots a_{k-1}, b = b_0 b_1 \dots b_{n-1}$
  $total \leftarrow 0$
  for $i \leftarrow n-1$ to 0 **do**
    $carry \leftarrow 0$
    $tmpAdd \leftarrow$ Array of $k+1$ digits
    for $j \leftarrow k-1$ downto 0 **do**
      $c \leftarrow b_i * a_j + carry$
      $tmpAdd_{j+1} \leftarrow c \mod 10$
      $carry \leftarrow \lfloor c/10 \rfloor$
    $tmpAdd_0 \leftarrow carry$
    $total \leftarrow total + tmpAdd * 10^{(n-i-1)}$
  return total

Recursive Split into 2 halves, $a = 10^{\lfloor k/2 \rfloor} l_a + r_a, b = 10^{\lfloor n/2 \rfloor} l_b + r_b$.
$ab = (10^{\lfloor k/2 \rfloor} l_a + r_a)(10^{\lfloor n/2 \rfloor} l_b + r_b) = r_a r_b + 10^{\lfloor n/2 \rfloor} r_a l_b + 10^{\lfloor k/s \rfloor} l_a r_b + 10^{\lfloor k/2 \rfloor + \lfloor n/2 \rfloor} l_a l_b$
Implement recursively, base case is single digit mult, if statements for $n > 1$ and $k > 1$ for term1, $k > 1$ for term2, $n > 1$ for term3.

Recursive Fast Same as recursive, but combine term3 and 4 into 1 multiplication.
$(l_a + r_a) * (10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor l_b + r_b} l_b + r_b) = 10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} l_a l_b + (l_a r_b + 10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} l_b + r_b) - 10^{\lfloor n/2 \rfloor - \lfloor k/2 \rfloor} l_a l_b - r_a r_b$
This is term3, and so we get:
$a * b = 10^{\lfloor k/2 \rfloor + \lfloor n/2 \rfloor} term2 + 10^{\lfloor k/2 \rfloor} term3 + term1$

3 Recursion

Algo is recursive if, while solving prob, calls itself 1+ times. Need a base case so recursion stops. Examples:

3.1 Recursive power computation

Algorithm power(a,n)
if n=0 **then** return 1
else
  $previous \leftarrow power(a, n-1)$
  return $previous * a$

3.2 Binary Search

Can implement through recursion. In: sorted array, start, stop, key. Out: index found or -1 if not found. Split in half, then recall binary search with corresponding half (depending on whether current index is larger or smaller than key) to check. Base case is when start=stop, check if value is key, else false.

3.3 Fibonacci Sequence

$F(0) = 0, F(1) = 1 \& F(n) = F(n-1) + F(n-2)$ if $n \geq 2$

Iterative
if $n = 0$ **then** return 0
if $n = 1$ **then** return 1
$previous \leftarrow 0$
$current \leftarrow 1$
for $i = 2$ to n **do**
  $tmpCurrent \leftarrow current$
  $current \leftarrow current + previous$
  $previous \leftarrow tmpCurrent$
return current

Recursive although still not efficient. $O(2^n)$
if $n = 0$ **then** return 0
else if $n = 1$ **then** return 1
else return Fib(n-1)+Fib(n-2)

4 Divide-and-Conquer

Many recursive algorithms: **Divide** prob into subprob, **conquer** subprob by solving them recursively, **combine** the subsols

4.1 MergeSort

$O(nlogn)$ Array to be sorted, divide in 2 halves, conquer by recursively sorting each half, then merge each half
To merge, create temp array with left and right indices, constantly comparing
Given 2 sorted halves, merge to one sorted array. left to mid sorted, mid+1 to right sorted.

Algorithm merge(A, left, mid, right)
$indexLeft \leftarrow left$ //Left half index
$indexRight \leftarrow mid + 1$ //Right half
$tmp \leftarrow$ Array of same type and size as A
$tmpIndex \leftarrow left$ // start at begin
while $tmpIndex \leq right$ **do** // Go up to right
  if $indexRight > right$ or ($indexLeft \leq mid$ and $A[indexLeft] \leq A[indexRight]$) **then** //indexRight is end or indexLeft isn't at mid yet and element there is smaller than at indRight (left smaller than right)
    $tmp[tmpIndex] \leftarrow A[indexLeft]$ // Take left & increment
    $indexLeft \leftarrow indexLeft + 1$
  else//Right isn't at end, right is smaller than left
    $tmp[tmpIndex] \leftarrow A[indexRight]$ // Take right
    $indexRight \leftarrow indexRight + 1$
  $tmpIndex \leftarrow tmpIndex + 1$
for $k \leftarrow left$ to $right$ **do** $A[k] \leftarrow tmp[k]$ // Copy tmp to A

mergeSort, keep splitting in half until you merge trivial, recursion

Algorithm mergeSort (A, left, right)
if $left < right$ **then** // At least 2 elements
  $mid \leftarrow \lfloor (left+right)/2 \rfloor$
  mergeSort($A, left, mid$)
  mergeSort($A, mid+1, right$)
  merge($A, left, mid, right$)

```
mergeSort([ 3 1 5 4 2 ], 0, 4)
  mergeSort([3 1 5 4 2], 0, 2)
    mergeSort([3 1 5 4 2], 0, 1)
      mergeSort([3 1 5 4 2], 0, 0)  // nothing to do
      mergeSort([3 1 5 4 2], 1, 1) // nothing to do
      merge([3 1 5 4 2],0,0,1)     //array becomes [1 3 5 4 2]
    mergeSort([1 3 5 4 2], 2, 2)   // nothing to do
    merge([1 3 5 4 2],0,1,2)       // array stays [1 3 5 4 2]
  mergeSort([1 3 5 4 2], 3, 4)
    mergeSort([1 3 5 4 2], 3, 3)   // nothing to do
    mergeSort([ 1 3 5 4 2], 4, 4)  // nothing to do
    merge([1 3 5 4 2], 3,3,4)      // array becomes [1 3 5 2 4]
  merge([1 3 5 2 4], 0, 2, 4)      // array becomes [1 2 3 4 5]
```

Something like $T(n) = 1 + 2T(n/2) + n$

4.2 QuickSort

$O(n^2)$, but usually faster than MergeSort, since $O(nlogn)$ on average. Not as reliable because of n^2. Divided and conquer again. Pick a pivot and put smaller things on left, bigger on right, then insert pivot in middle and use recursion.

Algorithm quickSort(A,start,stop)
if $start < stop$ **then**
  $pivotIndex \leftarrow partition(A, start, pivotIndex)$
  quickSort($A, start, pivotIndex - 1$)
  quickSort($A, pivotIndex + 1, stop$)

Worst case is when already sorted. Usually will split into roughly equal parts if random. Easier to do in-place than MergeSort.
partition, takes an array with indices and stop, out: return index j, rearranges all elements of A so all indexes below j are lower than A[j] and all above are greater than A[k]

$A = [6, 3, 5, 9, 2, 5, 7, 8, 4, 5_{\text{pivot}}]$

$\xrightarrow{\text{partition}} \underbrace{3, 5, 2, 5, 4,}_{\leq pivot} 5_{\text{pivot}} \underbrace{6, 9, 7, 8}_{\geq pivot}$

Quicksort each side now

$3, 5, 2, 5, 4_{pivot}$ $\quad 6, 9, 7, 8_{pivot}$

$\xrightarrow{\text{partition}} \underbrace{3, 2}_{\leq pivot}, 4, \underbrace{5, 5}_{\geq pivot} \quad \underbrace{6, 7}_{QS}, 8, \underbrace{9}_{QS}$

Quicksort again
$2, 3, 4, 5, 5, 5, 6, 7, 8, 9$

Algorithm partition(A,start,stop)
$pivot \leftarrow A[stop]$
$left \leftarrow start$
$right \leftarrow stop - 1$
while $left < right$ **do**
  while $left < right$ and $A[left] < pivot$ **do** $left \leftarrow left + 1$
  while $left < right$ and $A[right] \geq pivot$ **do** $right \leftarrow right - 1$
  //Basically keep indexing left and right until number on left and right don't belong, swap
  if $left < right$ **then** exchange $A[left] \leftrightarrow A[right]$
exchange $A[stop] \leftrightarrow A[left]$
return left

5 Loop invariants

Algo can be described by input, output, **preconditions** (restrictions on input), **postconditions**
Ex: Bin search, input, array of integers, output index, prec: array sorted in ascending, postc: index is in array, -1 if not
Check correctness of algo: for correct input data, **stops** and produces correct output, input satisfies prec, output satisfies postc. How to prove?
Loop Invariant loop property that holds before and after each iter of loop. To prove using LI, need 3 things:
Initialization: true before first iter of loop, **maintenance:** true before an iter and stays true before next iter, **termination** when loop terminates, invariant gives useful prop to show correct
Similar to induction, **base case**, **inductive step**. Invariant holds before first iter (base case), invariant holds from iter to iter (inductive step), termination is different, stops induction. Can show 3 in any order.

Example Insertion sort
Loop invariant: A[0...i-1] sorted. **Init** before i=1, A[0] sorted, **Maint** inserting ith element keeps A sorted **Term** outer for loop ends when i is length of A. Plug into i-1, get A[0...length-1], which is same as original Array, but sorted

FindMin In: array A of n int, Out: smallest
Algo FindMin(A, n)
$i \leftarrow 1$
$m \leftarrow A[0]$
while $i < n$ **do**
  if $A[i] < m$ **then** $m \leftarrow A[i]$
  $i \leftarrow i + 1$
return m

LI here is: at iter i, $m = min\{A[0], \dots, A[i-1]\}$
init: $i = 1, m = A[0] = min\{A[0]\}$
maint: Assume LI holds at begin, $m = min\{A[0], \dots, A[i-1]\}$
2 conditions, $A[i] < m$: replace $A[i]$ makes $m = min\{\dots A[i]\}$
$A[i] \geq m$, then don't change m and $m = min\{\dots A[i]\}$
term: Algo will stop because i will reach n. Loop stops when $i = n$, so by LI, $m = min\{A[0], \dots, A[n-1]\}$

6 Running time

Measure speed of algo. But, depends on **size** of input, so describe as function of input size.
Also depends on **content** of input, like, if sorted or not
3 possibilities, best case (usually meaningless), average case (hard to measure), worst case (good for safety critical & easier to estimate)

7 Primitive Operations

Ops that can be performed in constant time, assume they all take same time
$T_{assign}, T_{call}, T_{return}, T_{arith}, T_{comp}$ (compare), $T_{cond}, T_{index}, T_{ref}$ (follow obj ref)
To find func of running time, add all primitives, including things depending on n (loops)

8 Big-O

Simplify discussion of runtime, describe how running time is for LARGE n, grows as most fast as $O(g(n))$
$f(n)$ and $g(n)$ 2 non-negative funcs defined on $\mathbb{N}$ $f(n)$ is $O(g(n)) \iff \exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}, st. \forall n \geq n_0, f(n) \leq c \cdot g(n)$ c cannot depend on n
To prove $f(n)$ is $O(g(n))$, find n_0 and c to satisfy conditions. Manipulate inequalities.
To prove $f(n)$ is not $O(g(n))$, show for any n_0 and c, there's an $n \geq n_0$ st. $f(n) > cg(n)$ (usually n is in terms of c)

8.1 Hierarchy

$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n^k) \subset O(2^n)$
$O(1)$, functions bounded above by a constant.

8.2 Shortcuts

1. Sum rule. $f_1(n) \in O(g(n))$ & $f_2(n) \in O(g(n))$ then $f_1(n) + f_2(n) \in O(g(n))$ Can prove using 2

cs and 2 ns
2. **Constant factors rule** $f(n) \in O(g(n))$ then $kf(n) \in O(g(n))$ for any constant k.
3. **Product rule** $d(n) \in O(f(n))$ and $e(n) \in O(g(n))$ then $d(n) \cdot e(n) \in O(f(n) \cdot g(n))$
4. $n^x \in O(a^n)$ for fixed $x > 0$ and $a > 1$
5. $log(n^x) \in O(log(n))$ for fixed $x > 0$. Prove by $log(n^x) = xlog(n)$
6. $\log_a(n) \in O(\log_b(n))$, prove by dividing, $\log_a(n) = \log_b(n)/\log_b(a)$

Limits 1. $\lim_{n\to+\infty} f(n)/g(n) = 0 \implies f(n) \in O(g(n))\&g(n) \notin O(f(n))$
2. $\lim_{n\to+\infty} f(n)/g(n) = x \neq 0 \implies f(n) \in O(g(n))\&g(n) \in O(f(n))$
3. $\lim_{n\to+\infty} f(n)/g(n) = +\infty \implies g(n) \in O(f(n))\&f(n) \notin O(g(n))$
4. $\lim_{n\to+\infty} f(n)/g(n)$ does not exist, says nothing
Remember l'Hôpital's rule: $\lim_{n\to+\infty} f(n)/g(n) = \lim_{n\to+\infty} \frac{df(n)/dn}{dg(n)/dn}$

8.3 Big-Theta
$f(n)$ is $\Theta(g(n)) \iff f(n)$ is $O(g(n))$ and $g(n)$ is $\Theta(f(n))$

8.4 Big-Omega
$f(n)$ is $\Omega(g(n)) \iff g(n)$ is $O(f(n))$

9 Abstract Data Types
Model of a data structure that specifies type of data stored and operations supported on data.

Specifies what can be done with data, but not how it is done. Implementation of ADT specifies how operations are performed. User does not need to know about implementation.

9.1 List ADT
Stores an ordered set of objects of any kind.
Operations: **getFirst()** : returns first obj, **getLast()**: returns last object of list, **getNth(n)**: returns n-th obj, **insertFirst(Obj o)** : adds o at begin, **insertLast(obj o)**: adds o the end of the list, **insertNth(n,o)** : adds n-th object as o, **removeFirst()**: remove first obj, **removeLast()**: remove last o, **removeNth(n)**, **getSize()**: returns # of obj in list, **concatenate(List l)**: append l to end of this list

Implementation with an Array

1D array L to store elements, int size for # obj stored (not capacity)
getFirst() will return L[0], getLast() returns L[size-1] and getNth(n) returns L[n]
insertLast increments size and puts at last spot, but insertNth has to shift all elements by 1 and increment
removeLast decrease size by 1 (no need to del things)
removeNth shifts over nth, size-1
Arrays good sine easy to implement & space efficient
Limitations, size has to be known in advance, mem needed might be larger than num of elem used, insert or del can take $O(n)$. Array implementation is bad when # of objects not known in advance and/or lots of insertions or removals.

Implementation with a linked-list, sequence of nodes, store data and which node is next in list. Have **head** and **tail**. recursive data structure.
Good since don't need to know size, can expand and shrink easy, memory proportional to size

```
public class node{
private Object value;
private node next;
//Constructor
public node
(Object x, node n){
        value=x;
        next=n;}
public node getNext(){
return next;}
public Object getValue(){
return value;}
public void
setValue(Object x){
value=x;}
public void setNext(node n){
next=n;}
}

class linkedList{
node head, tail;
//default constr, empty list
list(){
        head=null;
        tail=null;}
getFirst(){
if(head==null) throw..
return head.getValue();}
getLast(){
if(tail==null) throw..
return tail.getValue();}
getNth(){
// throws outofbounds if...
node current=head;
while(n>0){
current=current.getNext();
n--;}
return current;}
void addLast(Object x){
if(tail==null){//empty list
tail=head=new node(x,null);}
else{
tail.setNext(new node(x,null));
tail=tail.getNext();}
}
void addFirst(Object x){
// less costly than Array
// O(1) here vs O(n)
head=new node(x,head);
if(tail==null) tai=head;}
insertNth(int n, Object x){
// throw if out of bounds
while(n>1){
predecessor=
predecessor.getNext();
n--;}
node newelem=
new node(x,predecessor.getNext());
predecessor.setNext(newelem);
return true;}
removeFirst(){
if(head==null){
return false;}
head=head.getNext();}
removeLast(){
if (tail==null){
return false;}
tmp=head.getNext();
while(tmp.getNext!=tail)
tmp=tmp.getNext}
tail=tmp;
tail.setNext(null);}
remove(Object x)
//throw if not found
if(head==null) //throw empty
if(head.getValue().equals(x)){
head=head.getNext();
if(head==null)tail=null;
return true;}
node current=head;
while(current.getNext()!=null &&
!current.getNext().
getValue().equals(x))
// right before x{
current=current.getNext();}
if(current.getNext()==
null) return false;
else{
current.setNext(current.getNext().
getNext());
if(current.getNext()==null){
tail=current;}
} return true;
}
```

9.2 Stacks
ADT list only allowing ops at one end of list (top)

Ops **push(obj)**:insert elm at top, **obj pop()**; removes obj at top; **obj top()**: return last inserted w/o remove, peek() in java, **size()**: # elem, boolean **isEmpty()**: empty?
Stack is a Last in - First out (LIFO)
Use: browser history, undo, chain of method calls in JVM
Method Stack in JVM consists of every method call, with local vars and return, etc. Allows recursion

```
public class ArrayStack{
        private Object S[];
        private int top=-1;
        public ArrayStack
        (int capacity){
        S=new Object[capacity]}
}
```

index t keeps track of top element

push(o)
  if $t = S.length - 1$ **then**
    throw FullStackEx
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$
size()
  return $t + 1$
pop()
  if isEmpty() **then**
    throw EmptyEx
  else $t \leftarrow t - 1$

Perf: $O(n)$ space, ops take $O(1)$. Limits: Need max size, push into full gives exception
Can use Singly Linked List instead
top element is stored first node
Space used $O(n)$ and each operation takes $O(1)$
Can use stacks to check if parenthesis match, put opening bracket on stack and remove if it finds a match, valid if stack is empty at end

9.3 Queues
First in first out data structure, first come first serve service

Ops void **enqueue(obj o)**: add o to read, obj **dequeue()**: remove obj at front, exc if not, obj **front()**: returns obj at front, doesn't remove, exc if empty, int **size()**: return #, boolean **isEmpty()**:empty?

Implement with linked-list
enqueue>addLast; dequeue>removeFirst; front>getFirst; empty>empty; size>size
All $O(1)$ except size $O(n)$

Double-ended queues deque, allows insertion+removal from front and back
Implement with linked list:
getFirst(),getLast(); addFirst(o);addLast(o); isEmpty(); removeFirst(); all $O(1)$
removeLast(); size(); $O(n)$
Problem: removeLast takes $O(n)$
To do it faster: doubly-linked-list, have ref to prev too

```
class node{
        node prev,next;
        Object value;
        node(Object val,
        node p, node n);
        node get Prev();
        void SetPrev(node n);
        node getNext();
        void SetNext(node n);
        Object getValue();
        void setValue(Object o);}
```

Now removeLast(); can be done in $O(1)$.

Deques with Arrays If we know deque will never have more than N elements. Keep indices for head & tail.

```
addLast(o){tail=tail+1;
L[tail=o]}
addFirst(o){head=head-1;
L[head=o];}
removeLast{tail=tail -1;}
removeFirst{head=head+1;}
```

Adding just increments head ref by one, doesn't shift because too costly.

Rotating arrays Avoid outOfBounds exceptions, wrap around. Take $a \mod N$, where N is size of array. Deque will never go out of bounds, but can overwrite itself, so check if full when adding. Initialize head and tail at -1. Need to handle: only one object to remove, inserting first element and isEmpty/isFull.

10 Extra Java Stuff
Inheritance, new object inherits data properties from parent, can add extra ones. Same for methods, although can overwrite some.
public class HockeyTeam **extends SportsTeam**
do-while loops, checks condition after executing

```
do {
...
} while (condition);
```

File-IO, remember to import java.io.*
Read from keyboard

```
BufferedReader kb
= new BufferedReader
(new InputStreamReader(System.in));
String name = keyboard.readLine();
keyboard.close();
```

Also

```
Scanner reader
= new Scanner(System.in);
wordUntilSpace=reader.next();
.nextDouble, .nextInt, etc.
reader.close();
```

File reading, **checked exception**, need to catch IOException or throws FileNotFoundException in header

```
Scanner fileRdr
= new Scanner (new file("foo.txt"))
BufferedReader br
= new BufferedReader (fileRdr);
br.readLine();
FileWriter fw
= new FileWriter("foo.txt");
BufferedWriter bw
= new BufferedWriter(fw);
bw.write("Hi"); bw.newLine();
bw.close(); fw.close();
```

To read from URL

```
URL mcgill=new URL("www...");
URLConnection mcgillConn
=Mcgill.openConnection();
BufferedReader myURL =
new BufferedReader(
new InputStream Reader
(mcgillConn.getInputSteam()));
// readLine, etc
```

Throwing throw new IllegalArgumentException("RIP")

11 Problems
List-intersection problem: input (names of students in COMP250 and names of students in MATH240, no one with same name)
How many are taking both classes? Minimize times to compare?
Can nest for-loops → inefficient
Binary search

listIntersection (A,m,B,n)
  inter ← 0
  B ← sort(B,n)
  for i ← 0 to $m - 1$ **do**
    if binarySearch(B,n,A[i]) **then**
      inter ← inter+1
  return inter

For actual binary search, see algos

12 Formulas/Math
$\sum_{k=0}^{n-1} ar^k = a\frac{1-r^n}{1-r}$ for $r \neq 1$
$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

12.1 Logarithms
- $y = \log_a(x) \iff a^y = x$
- $\log_b(mn) = \log_b(m) + \log_b(n)$
- $\log_b(m/n) = \log_b(m) - \log_b(n)$
- $\log_b(m^n) = n \cdot \log_b(m)$

12.2 Induction
Base case, induction step using induction hypothesis.

12.3 Recurrence
Get an explicit formula for a recursive formula by using back-substitution.