1 Algorithms

An algorithm is a **systematic** and **unambiguous** procedure producing an answer to a question/a solution in finite **number of steps**.

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

In-place algorithms Uses constant amount of memory (+that used to store input). Important, if data barely fits mem, don't want to use 2x memory.
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 ← $\lfloor(left + right)/2\rfloor$
  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 $ct \leftarrow 0$ 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)

3	3	3	3	3	3
17	17	-5	-5	-5	-5
-5	-5	17	-2	-2	-2
-2	-2	-2	17	17	17
23	23	23	23	23	4
4	4	4	4	4	23

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($list[i], list[tmpIndex]$)

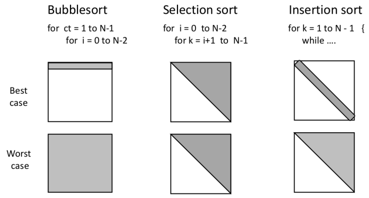
3	-5	-5	-5	-5	-5
17	17	-2	-2	-2	-2
-5	3	3	3	3	3
-2	-2	17	17	4	4
23	23	23	23	23	17
4	4	4	4	17	23

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
    $list[i] \leftarrow list[i-1]$
    $i \leftarrow i-1$
  $list[i] \leftarrow elementK$

-5	-5
3	-2
17	3
-2	17
23	23
4	4



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. Divide 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**
  $pivotIndex \leftarrow partition(A, start, stop)$
  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

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

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

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], $O(1)$
insertLast increments size and puts at last spot, $O(1)$, but insertNth has to shift all elements by 1 and increment, $O(n)$
removeLast decrease size by 1 (no need to del things) $O(1)$
removeNth shifts over nth, size-1, $O(n)$
Arrays good since 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
getFirst $O(1)$, getLast $O(1)$, getNth $O(n)$, insertLast/insertNth $O(n)$, removeLast $O(n)$, removeNth $O(n)$

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

Array-based Stack 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 end, 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 & removeLast $O(n)$

Double-ended queues deque, allows insertion+removal from front and back
To do it faster: doubly-linked-list, have ref to prev too 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.

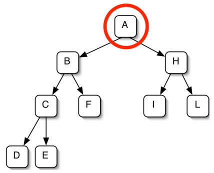
10 Trees

treeNode ADT, has object value and 3 treeNodes, parent and 2 children. Some may be null. Operations: **getValue(); getParent(); getLeftChild/RightChild/-Sibling(); setParent/LeftChild/RightChild(treeNode n) depth(); height();**
Root: only node with null parent. **Siblings(X)**, nodes with same parent as X, not counting X. **Descendants(X)**, nodes below X. **Ancestors(X)**, nodes between X and root. No children = **leaves**. Nodes with children =**internal nodes**. Tree is **ordered** if order of children of a node matter.
Depth(x), number of ancestors of x. **Height(x)**, number of nodes in longest path of x to leaf (exclude x). Height of tree is height of root.
Applications: data storage, compression, job scheduling, pattern matching, compilers, biology, decision trees, math expressions (nodes are ops, leaves are vals), parse tree for phrase structure

10.1 Binary Trees

Every node, at most 2 children (left & right). **Proper binary tree**, every internal node has 2 children.

10.2 Tree Traversal
To visit all nodes of a tree from root, use recursion.




```
preorderTraversal(treeNode x)
print x.value
for each c in children(x) do
    preorderTraversal(c)
```
A B C D E F H I L. Go left calling preorder, then go right calling preorder

```
postorderTraversal(treeNode x)
for each c in children(x) do
    postorderTraversal(c)
print x.value
```
D E C F B I L H A. Left, right and then node itself

```
inorderTraversal(treeNode x)
inorderTraversal(x.leftChild)
print x.value
inorderTraversal(x.rightChild)
```
D C E B F A I H L. Left subtree, node then right

10.3 Dictionary ADT
Map, stores pairs (key,value). Data accessed through key: **obj find(key); insert(key,object); obj remove(key)**. If keys can be ordered, we also have **obj previous/next(key);**

Array Implementation Array of pairs (key,val). Find(scan whole array), remove(find and then shift) are $O(n)$. insert is $O(1)$, just have to add pair at end and size++ if sorted by key: find (bin search)$O(\log n)$, (bin search, shift, insert)insert is $O(n)$, remove is same, except with bin search

Linked-list Implementation Each node has pair. find and remove are $O(n)$, insert is $O(1)$.

10.4 Binary Search Tree
Binary tree, with elements on left having key $\leq$ and elements on right $\geq$

Find Start from root, choose left or right until you find key or get to leaf. If node is null, return null (false), if node's key is k, return node, else check if key is > k or not, greater, recurse on left child, smaller, recurse on right child.

Insert Go down tree like in find, add new child (left or right, depends on key)

Remove Find node to remove using find. If leaf, remove. If internal node with one child, replace by child. If two children, replace by node with largest key smaller than key to remove. Go right once, and keep going left.

10.5 Hash Tables
Dictionary. Keys are integers between 0 and $K-1$. Use $A[0\ldots K-1]$ to store dictionary. insert makes key index equal to val, remove makes it null, find returns that index. All operations are $O(1)$, but takes up A LOT of memory if K is large.

Hash Functions Map K keys to N integers, N much smaller than K. $f:[0\ldots K-1]\to[0\ldots N-1]$, get $f(k)$ as index for ops. Collisions, many keys map to same index. Solution: each element of array is a dictionary (bucket), implemented via linked-list, BST or hash table. Chaining: insert becomes $A[f(k)].insert(k,i)$;, remove $A[f(k)].remove(k)$;, find $A[f(k)].find(k)$;

Runtime of chaining: Compute hash func: $O(1)$. Insert $O(1)$. Search = hash func + search, worst, all keys go in same bucket, $O(n)$. Deletion, $O(1)$+search

Want to spread out hash function evenly among buckets. Choice depends on application, in general, $f(k)=k \pmod N$ good if N prime. If key is not an integer (String), map key to int first and then hash function. Ex. map to sum of ASCII (collisions, bad), or choose prime and multiply each letter by p^{index} and sum.

11 Priority Queue ADT
Like a dictionary, stores pairs. Rank of object depends on priority (key). Lower key $\Longrightarrow$ front of queue. **findMin(); removeMin(); insert(key, obj);**. Applications: customers in line, compression, AI, graph search.

Unsorted array: findMin scans array $O(n)$, insert puts object at end $O(1)$, removeMin finds min then shifts $O(n)$.

Sorted array: findMin returns first element $O(1)$, insert uses BST to find position, then shifts, $O(n)$, removeMin removes first elem, but have to shift

Sorted doubly-linked list: findMin, first elem $O(1)$. Insert has to scan $O(n)$, removeMin removes first elem $O(1)$

11.1 Heap ADT
findMin, $O(1)$, removeMin() $O(\log n)$, insert $O(\log n)$. Heap is based on binary tree but not binary search tree. For any node other than root, key is larger than parent's key. All but second to last levels are full, last level is packed left. For $i=0,\ldots,h-1$, 2^i nodes of depth i. Numbers increase going left to right, 1 horizontal layer at a time. Height of heap: max number of nodes in heap of height h: $\sum_{k=0}^{h}2^k=2^{h+1}-1$, min number is $(2^h-1)+1=2^h$, so height of heap is $\lfloor\log n\rfloor$. findMin(), min is always root. Insert: two steps, find leftmost (on last row) unoccupied node, insert, have to then restore order.

Bubbling-up: to restore, keep swapping with parent as long as key is smaller than parent, $O(\log n)$. RemoveMin(), replace root with last node (left most, last row) and then restore heap-order. Bubbling-down: to restore, keep swapping node with smallest child as long as node is larger than child's key, $O(\log n)$

To find where to insert (most left), given the last node, keep going up while current node is right child. If parent is then null, (root), insert at left child of node all the way on the left. Else, go up one more level, insert at missing right child (if there's already a right child, go down right, then keep going left until nothing). $O(n)$

Array representation of heaps: n keys, array of size $n+1$. Node at index i has parent at index $\lfloor i/2\rfloor$, left child is $2i$, right child is $2i+1$ with last node being first empty cell. Add or subtract one to update.

```
heapSort(array A[0...n-1])
Heap h ← new Heap()
for i = 0 to n-1 do
    h.insert(A[i])
for i = 0 to n-1 do
    A[i] ← h.removeMin()
```
$O(n\log n)$, can do in-place by using array to store heap.

12 Graphs
Graph is pair (V,E), with V being set of nodes called **vertices** and E being pairs of vertices, **edges**.

Edge types: **Directed edge**, ordered pair (u,v), u is origin, v is destination. **Undirected edge**, unordered. **Directed graph**, all edges directed. **Weighted edge**, edge has real number associated to it (like distance). **Weighted graph**, all edges weighted.

Labeled graphs Vertices have names, geometric layout doesn't matter, connections do. **Unlabelled graph**, no names

Terminology **Endpoints of an edge**, 2 vertices at end. **Edges incident on vertex**, have vertex as endpoint. **Adjacent vertices** connected by edge. **Degree of vertex**, # incident edges. **Parallel edges** have same endpoints, multi edge graph counts both for degree. **Self-loop**, edge from vertex to self. **Path** sequence of adjacent vertices. **Simple path** all vertices distinct. Graph is **connected** $\Longleftrightarrow\forall$ pairs of vertices, $\exists$ path between them. Have to take into account directions. **Cycle**, path starts and ends at same vertex. **Simple Cycle**, all vertices distinct. Tree is **connected acyclic**, no cycles

For undirected graphs, each edge contributes to deg of 2 nodes: $\sum_{v\in V}\deg v=2|E|$

Undirected graph with no self-loops or multiple edges, $|E|\leq|V|(|V|-1)/2$

Implementing Graphs - Adjacency Lists Graphs can be stored as dictionary, with key = vertex identifier, info containing list of adjacent vertices. Using linked-list, we account for every adjacent vertex twice, redundant as we have to search through both lists. Ops: **addVertex(key)**, inserts entry into dict. **addEdge(key,key)**, inserts opposite vertex to both lists. **areAdjacent** Call find on opposite vertex

Adjacency matrix Decide order of vertices. Store as $n\times n$ array of boolean, $M[i][j]$, 1 if edge between i and j, 0 otherwise. Ops: **addEdge(i,j)**; $m[i][j]=1$, **removeEdge(i,j)**; $m[i][j]=0$. Not good for inserting/removign vertices, requires shifting. Needs space $O(n^2)$. Not good for parallel edges, works with weighted, can change 1 to weight.

List vs mat Lists better for: frequently add/rem vert, few edges, need to traverse. Mat better if frequently need to add/remove edges, not vertices. Check for edges. Matrix small enough to fit in memory.

12.1 Graph Traversals
Need to visit each node in a graph given one vertex A and method getNeighbors(vertex v).

Applications Exploration of graph not known/too big, web crawling, maze. Graph can be computed as you go along, game strategy, rubik's cube.

12.2 Depth-First Search
Go deep, keep visiting unvisited neighbors, go back at dead end. Maze: mark intersection, corridors and dead ends. Mark corridors (edges) as visited, keep track of path back. Rubik's cube, vertices are cube configurations, edges are configs one rotation away from each other. Can check if graph is connected, if graph has cycles.

```
DFS(v)
v.setLabel(VISITED)
for all u ∈ v.getNeighbors() do
    if u.getLabel()! =VISITED then
        DFS(u)
```
DFS called once for every vertex, for loop runs deg(v) # times, so run time is $2|E|\Longrightarrow O(|E|)$.

12.3 Breadth-First Search
Explore by layers. Explore all neighbors of v, then all neighbors of these neighbors... Can use to find shortest path.

```
iterativeBFS(v)
q ← new Queue()
v.setLabel(VISITED)
q.enqueue(v)
while (!q.empty()) do
    w ← s.deque()
    for all u ∈ w.getNeighbors() do
        if u.getLabel()! =VISITED then
            u.setLabel(VISITED)
            s.enqueue(u)
```
Also $O(|E|)$. Iterative DFS is the same, except it uses a stack.

12.4 Graph Problems
Shortest path **Unweighted:** Find min # edges between u & v. Algo: Do BFS from u to v, keep track of path length. **Weighted:** Find min total edge weight from u to v. Visit vertices in increasing order of distance from u, first time you get to v is shortest path. Can use priority queue. Apps: Going from 1 city to another, route packets through internet, solve puzzle in least # moves

Eulerian Cycles Visits each edge exactly once (vert can be visited mult times), for undirected graph. Find eulerian cycle if exists. Start at a vertex, follow unvisited edge (as long as does not result in graph with unvisited edge that is unreachable). Fast algo, no planning ahead

Hamiltonian cycle Visits each vertex once. Undirected graph, find hamiltonian if exists. Algo, very hard, try all possible $(n-1)!$ orderings. No polynomial algo

Graph coloring Undir graph, find min number colors needed to paint vert so no pair of adjacent have same color. App: color maps. No poly algo

Cliques Undir graph, clique is subset of vertices where all vertices adj. Find largest clique. No poly algo

Matching Pair vertices, try to match everybody. App: marrying

13 Dynamic Programming
Recursive algos can be slow if they have to recompute same numbers over and over (ex. Fibonacci). Div & conquer is top-down approach. Dynamic programming is bottom-up approach, use sols of small probs to get sols of larger probs. Store all fib numbers in an array, calculate in a loop. Fib becomes $O(n)$ instead of exp.

Change making prob Smallest number of coins needed to make x cents, given there are cents of val $C_1,C_2,\ldots,C_k$. Recursive algo for $opt(n)$: $opt(0)=0, opt(n)=1+\min\{opt(n-C_1),\ldots,opt(n-C_k)\}$. Recomputes like fib. Dyn prog: Use same formula, but iterate from bot to top to calc.

```
makeChange(C[0...k-1], n)
int X[] ← new int[n+1]
X[0] ← 0
for i ← 1 to n do
    smallest ← ∞ // get min using this loop
    for j ← 0 to k-1 do
        if C[j] ≤ i then
            smallest ← min(smallest, X[i-C[j]])
    X[i] ← 1 + smallest
Return X[n]
```
Greedy approach Choose what brings you closest to goal. Take as many highest denomination coins as possible then second highest ... Not always optimal. Problem has **greedy choice property** if optimal sol can be reached by greedy choices. Usually not optimal for optimization problems, but when they are, usually fastest.

Longest Increasing Subsequence Given array of integers. Slow algo: try all possible subseq. Dyn algo: $LIS[i]$ =len of LIS ending at index i and containing $A[i]$

```
LongestIncreasingSubsequence(A,n)
LIS[0] = 1
for i ← 1 to n-1 do
    LIS[i] ← -1 // for ineq
    for j ← 0 to i-1 do
        if (A[j] < A[i]&&LIS[i] < LIS[j]+1) then
            LIS[i] ← LIS[j]+1
return max(LIS)
```
Dyn algos mainly for optimization. Need properties to use dyn algos: **simple subprobs** be able to break into subprobs, **subprob optmization** optimal sol of big prob must be combination of opt sol to subprobs, **subprob overlap** opt sol to unrelated probs contain subprobs in common

14 Heuristics
Lots of important problems **NP-Complete**, probably no poly-time algo exists to guarantee good ans. Give algo that might give decent answer. **Heuristic algorithms** have no guarantee of producing right answer, tend to work well. Used for difficult opt probs.

Traveling Salesperson Problem Given set of n cities to be visited, distance matrix D with distances between cities. Want to visit each city exactly once and terturn to starting point, minimize total distance. Decision prob vers is NP-Complete. Suboptimal sols: Greedy algo(is a heuristic): Start at randomly chosen, move to closest unvisited. Or: choose pair of cities (edges of your graph) that are closest, as long as they don't close cycle (except last one)

Fastest/gradient descent heuristics Start with random sol S, consider neighborhood set of solutions, replace S by sol with best score. **Neighborhood** can be many things: changing pos of one city, exchanging pos of two cities, reverse order in which consecutive cities visited. Larger neighbor $\rightarrow$ higher chance of getting good sol, but more time to eval each neighbor. Often get stuck in local optima sol, sol that is not optimal but has no neighbors that are better. Good thing to do fastest descent multiple times and take best result. To avoid: try randomizing choice a bit, higher prob for good neighbors, but still small prob for bad neighbors

15 Extra Java Stuff
15.1 Java Collections
Interfaces Similar to class, but only has method signatures. Does not implement anything, just has method headers. Can implement with a class. Generic interface with <T> as type of object stored.

interface List<T>

A class can implement an interface, has code for every method, can have extra methods.

class ArrayList<T> implements List

Instantiate generic class by ArrayList<String>myList = new ArrayList<String>();

Iterator Interface Used to traverse objects, boolean hasNext(); and next(); methods. To use: given obj called list that implements iterator, do Iterator<T>itr=list.iterator();

Enhanced looping: for(String s : list)

Iterable interface, means class has iterator method

Comparable Interface Has compareTo, returns 0 if equal, positive if this>other, negative else

public static <T implementsComparable> T max(Collection<T> coll)

Method header that operates on type that implements comparable, returns type T, argument is a collection of objects of T

Can have type as <Integer,String> to store a tuple.

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")

16 Misc
16.1 Search Engines
Web crawling Use DFS or BFS to learn structure of graph, build index of web, hash table for word+list of sites. i.e. if site contains Java, add to Java entry

Idea 1: Pages should contain query words. Use index. Better to have several occurences of word in query. Allow synonyms, use context to determine meaning of words. Rough approx, can easily fool.

Idea 2: Look at graph structure. Web authors know good sites, link them. Good sites (authorities) are cited by many other sites, prefer sites with large in-degree

Idea 3: Site slinking to large number of sites (hubs) are less valuable refs

Idea 4: Sites that are authorities are more valuable refs

Put idea 2,3 & 4 together. Page-rank of vertex v describes how authorative. Not based on query. To ans query, get all sites with words of query, sort in decreasing pr. $PR(v)$ page rank of v, $C(v)$ is out-deg of v. w_i link to v. Damping factor d for technical reasons

$$PR(v)=(1-d)+d*\left(\frac{PR(w_1)}{C(w_1)}+\ldots+\frac{PR(w_k)}{C(w_k)}\right)$$

To solve for PR(v), we have system of linear equations. Gaussian elimination would take $O(n^3)$. Use numerical approx instead

Fixed-point iterative solution Assign each $PR(v)$ a val until convergence, i.e. until they barely change. iter #0, make all PRs= 1. iter # 1, calc PRs given vals of #0 (doesn't matter if you use current iter vals or prev, will conv to same)

16.2 Game Strategy
One player games **8 queens**, place 8 queens on chess board, no queens atk each other. Brute force: Try all combinations (way too long). **Backtracking:** place queens from first row to last, when invalid board reached, go back to last valid board. Use recursion. Place queen in spot, if valid, call algo again, else, set spot to 0, ++ index

Two player games **Game trees**, tree of possible decisions for each turn and then decisions after that turn ...

Winning position pos s.t. if X plays optimally, X wins even if O plays opt (recursive: P is winning if P is immediate win/leaf or $\exists$ move leading to win pos). **Losing pos** O wins if plays opt, recursive def opposite of winning. **Tie**, recursive def, immediate tie or nothing leads to win but $\exists$ move $\rightarrow$ tie

Calculate pos on tree recursively. Go to leaves: for current player: win= **+1**, tie= **0**, loss= **−1**. Then go up and assign val of node to best choice for whoever's turn it is. **minimax principle**, Deciding move for X, return max among deciding moves of next layer for O. Deciding move for O, return min among deciding moves of next layer for X. $(+1/-1$ thing just mentioned, except can have larger vals, estimate potential below)

Some game trees too big, only look K moves ahead, estimate potential.

16.3 Graphics & Ray Tracing
Primite objects: polygons, spheres, cones. **Complex objects**: mesh of triangles, more triangles for more precision. **Ray-tracing Algo** Have world, set of $3D$ objects and pos of eye. Output is image, with pixels colored. For every pixel, trace ray from eye to pixel, first object we hit(intersect) is color we want. Recursive ray-tracing does ray-tracing from the object it hits as well, to get all the objects.

Finding intersections: calculate closest intersection quickly, store objects in data structure that lets you quickly discard objects that can't intersect. **Quad trees**,2D, divide world into 4 quadrants, keep subdividing if more than 1 obj per square. 3D world, eight octants. Use tree with children as quadrants. Find which main quadrant intersected, then find subquadrant intersected. Test intersection with leaf until found.

16.4 Cryptography
Alice wants to send secret message to Bob, no safe communication channel. Want to make sure if someone intercepts, can't understand. Apps: military, eCommerce.

Secret-key Encryption Alice uses secret algo to encrypt, Bob knows algo, can invert after receiving message. Ex. Caesar cypher, shift each letter by constant. Easy to break, alternative: substitution cypher, map letter to other letter. Frequency attack, look for most common letters, probably E, T, etc., pairs of letters. Sol: Change permutation often. Problem: Alice & Bob need to share without anyone knowing, if they can't comm safely, how do they agree? **Public-key cryptography** don't need to agree on a key. Bob has public key visible to all, ppl who want to send msg to Bob will use it to encrypt msg. Bob has secret key that no one knows about. Without secret key, hard to decrypt, so only Bob can recover with ease.

RSA Bob chooses two large primes p,q. public key $e=p\times q$. Let $\phi=(p-1)(q-1)$. Private key d, s.t $3d \pmod\phi)=1$. Encrypt via $encr(M)=M^3 \pmod e)$. Decrypt via $encr(M)^d \pmod e)$. No poly algo to factorize large integers. To compute mod of large numbers quickly, split exponent into powers of 2.

17 Formulas/Math
$\sum_{k=0}^{n-10}ar^k=a\frac{1-r^n}{1-r}$ for $r\neq1$

$\sum_{k=1}^{n}k=\frac{n(n+1)}{2}$

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

17.2 Induction
Base case, induction step using induction hypothesis.

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

17.4 Binary
Each pos from 0 to n is 2^i. Conv from dec to bin: div number by 2, writing remainder, going top to bot. Bin number is remainders read bot to top, placed left to right. Need $\lfloor\log_2 N+1\rfloor$ bits to rep N in bin