Max (Min) Heap Property: the key of a node is ≥ (≤) than the keys of its children
parent(i) = math.floor(i/2) left(i) = 2i right(i) = 2i+1
Height of a binary heap: log base 2 (n)

Max_heapify(A,i): correct a single violation of the heap property occurring at the root of a subtree in O(log n)
Assume trees rooted at left(i) and right(i(are max-heaps, but A[i] violates the max-heap property. ie A[i] smaller than at least one of A[left(i)] or A[right(i)]
1. Find index of largest element among these 3
2. If index != i, exchange A[i] with largest element then recurse on subtree (max_heapify(A,largest_index))

Build_Max_Heap(A):
Index $\lfloor n/2 \rfloor$+1 onwards are leaves of the tree (cus 2i>n for all $i \geq \left\lfloor \frac{n}{2} \right\rfloor + 1$)
Heap property may only be violated at nodes 1… $\lfloor n/2 \rfloor$ of the tree.
for i <- floor(length[A]/2) downto 1:
do Max_heapify(A,i) T(n)=O(n)

Heap Sort
1. Build Max Heap from unordered array, find maximum element A[1];
2. Swap elements A[n] and A[1]: now max element is at the end of the array!
3. Discard node n from heap(by decrementing heap-size variable)
4. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.
6. Go to step
After n iterations, heap is empty. Every iteration involves a swap and a heapify > O(logn) **Overall: O(nlogn)**

Insert(A,k) -> O(log n)
Add a node at the end and then use max_heapify.
heap_size(A) = heap_size(A) +1
A[heap_size(A)] = - infinity // new element just created
increase_key(A,heap_size(A),k)
Extract Max -> O(log n)
Remove max and replace with a leaf, then max heapify
Increase key -> O(log n)
start from bottom, search up tree, found key, increase its value.
if k<A[i] => error
A[i] = k
while i>1 and A[parent(i)<A[i] // violate max heap condition (child>parent)
exchange A[i] <-> A[parent(i)]
i = Parent(i) // go up the tree

Questions
Major disadvantage of merge sort is that it cannot easily be converted into a stable sort.
> False. Merge sort is stable whenever the underlying merge operation always breaks ties by selecting from the left sub-array, and it is easy to ensure that this is the case.

Profiling gives the number of times each subroutine is called, and the length of time required per call, and for all the calls. This can be used to determine which data structure executes a function more quickly. cProfile doesn't say anything about space usage, return types, or correctness.

log(n^2) = 2 log(n), which is θ(log n). However, log(2^n) is θ(n)
f(n)+g(n) = θ (max(f(n), g(n))
To show exponential growth dominate polynomial growth, show that (n^x)/(y^n) = 0 as n → ∞

Which of the two algorithms {HeapSort, MergeSort} is a better choice if space (memory usage) is the primary concern,
rather than running time?
> Heapsort; it sorts "in-place". Merge sort, as described in class, requires additional arrays to be allocated. (As a side note, using methods not described in class, in-place merging is possible, but it requires a great deal of work and is not at all practical).

- Heap sort, Insertion sort and Merge sort are stable algorithms.
- Insertion and Heap sort are in-place algorithms.
- For all values of n, there is an array of n distinct elements which this algorithm sorts in O(n) ->Only Insertion sort. (works in O(n) if elements are already sorted.)
- Heap, Insertion and Merge sort works in the comparsion model (only manipulation of elements is by pairwise comparisons)

Is it always true that a real-valued function f satisfies $f(n) = O(f(n)^2)$?
> False. Let f(n) = 1/n.
If $f(n) = \theta(g(n))$ and $g(n) = \theta(h(n))$, $then$ $h(n) = \theta(f(n))$. **>** True. Θ is transitive.
If f(n) = O(g(n)) and g(n) = O(f(n)), then f(n) = g(n).
> False. Let f(n) = n, g(n) = n+1.

If we were to extend our O(n) 2D peak finding algorithm to four dimensions, it would take O(n3) time.
Running merge sort on an array of size n which is already correctly sorted takes O(n) time. **>** False. The merge sort algorithm presented in class always divides and merges the array O(log n) times, so the running time is always O(n log n).
The height of any binary search tree with n nodes is O(log n).
> False. In the best case, the height of a BST is O(log n) if it is balanced. In the worst case, however, it can be θ(n).
The depths of any two leaves in a max heap differ by at most 1.
> True. A heap is derived from an array and new levels to a heap are only added once the leaf level is already full. As a result, a heap's leaves are only found in the bottom two levels of the heap and thus the maximum difference b/w any 2 leaves' depths is 1.
Insertion Sort for sorted list: running time is θ(n). Iterates over the list and for each element, swaps the element backwards till it's in the correct position. Thus, for a sorted list, each element will be swapped 0 times and run time is θ(n). Merge Sort: runtime always O(nlogn)

Suppose you are given a list of N integers. All but one of the integers are sorted in numerical order. Identify a sorting algorithm from class which will sort this special case in O(N) time. **>** Insertion sort. O(1) swaps for all but 1 element -> O(N) swaps. Total runtime is O(2N)=O(N).

$\lg(\sqrt{n}) = \theta(\lg n^3)$ (lg n)/n = Ω(1/n) $n\ log_{30} n = \theta(n \lg n)$
$3^n = \Omega(2^n)$ $\sqrt{2^n} = \Omega(n \lg n)$ $n^2 \leq n^{3+\sin(n)} \leq n^4$
$nlg\ n = O(n^{\frac{101}{100}})$ $n^{1.5} = \Omega(n \lg n)$ (since sin(n) b/w -1 & 1)
$1 = \theta(2 + \sin n)$ n! = O(n+1)! $n^{\log n} > n^c$
1 = Ω(lg n/n)

Complexity

- Input space: set of possible inputs (eg {expo, changi, simei})
- Input instance: particular input of a problem instance (eg x=expo)
- Output space: set of possible output (eg any positive number)

Asymptotic Complexity

- f(x) = Θ(g(x)) grows asymptotically equal EG. $x^2 = \theta(x^2 + x(lgx)^2)$
(⇔ ∃ D1,D2 > 0, n0 s.t. D1|g(n)| ≥ |f(n)| ≥ D2|g(n)| for n ≥ n0)
EG for $f(n) = 2n^2 + 100$, $g(n) = n^2$ let D1=2, D2=1.
- f(x) = O(g(x)) grows asymptotically <=
f grows at most as fast as g when x→∞ EG. n^2=O(n^10)
(⇔ ∃ D>0, n0 s.t. |f(n)| ≤ D|g(n)| for n ≥ n0)
- f(x) = Ω(g(x)) grows asymptotically >=
f grows at least as fast as g when x→∞
(⇔ ∃D>0, n0 s.t. |f(n)| ≥ D|g(n)| for n ≥ n0

Document Distance

```
For (i=0;i<n;i++)
   For (j=0;j<n;j++)
      Print("GREAT");
For (k=0;k<n;k++)
   print("GREAT");
```

$T(n) = n^2 + n$

$\theta(n^2)$

L is a List, D is a dictionary

Commands	Complexity
$D[x_i] = y_i$	Θ(1)
x in D	Θ(1)

Commands	Complexity
L[i] = x	Θ(1)
L.append(x)	Θ(1)
L1.extend(L2)	Θ(|L1|)
A = L1 + L2	Θ(|L1|+|L2|)
x in L	Θ(|L|)

Doc D: a seq of words
Word w: a seq of char
Word freq D(w): no. of occurrences of w in D
Vector Space Model: Treat each document as a vector of its words
Dot product & normalise result by measuring distance by angle b/w vectors.

If θ=0 documents "identical"
(if same size, permutations of each other)
if θ=π/2 don't even share a word
Algorithm: 1. Read File & make word list (divide file into words)
2. Count frequencies of words
3. Compute dot product (for each word in 1st doc, check if it appears in the other document. If yes, multiply their freqs and add to the dot product. Worst case time: order of #words(D1) x #words(D2)
If sort doc into word order, compute dot product in #words(D1) + #words(D2)

Insertion sort:

```
For i in range(1,n):
   while A[i]<A[i-1]:
      swap A[i] with A[i-1]
      i -= 1
```

Worst-case: T(n) = i comparisons & swaps at step i

$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \theta(n^2)$$

Divide & Conquer Solution:

Divide input into smaller problems, conquer each part recursively, combine results to obtain solution of original.

Merge Sort:

- if n=1: done –> return
- recursively sort A[:n/2] -> L
- recursively sort A[n/2:] -> R
- merge L & R -> output A'

Time:

1. Divide: Θ(n)
2. Recursion: n1=n2=n/2
time = T(n1)+T(n2)=2T(n/2)
3. Merge: Θ(n)
Total: T(n)=2T(n/2)+Θ(n)=Θ(*n*log*n*)

Master Theorem:

The Master Theorem applies to recurrences of the following form: T(n) = aT(n/b) + f(n) where a ≥ 1 and b > 1 are constants and f(n) is an asymptotically positive function.

h = # of levels = $\log_b n = \theta(\log n)$

L = # of leaves = $n^{\log_b a}$

a = # segments divided
n/b = # elements in sub-segments

There are 3 cases:

1. If $f(n) = O(n^{log_b a - \epsilon})$ for some constant ∈>0, then $T(n) = \theta(L)$ (geometrically ↑ing down the tree)
2. If $f(n) = \vartheta(L) \Rightarrow T(n) = \vartheta(L \log n)$ (roughly equal at each lvl)
3. If $f(n) = \Omega(L^{1+\epsilon}) \Rightarrow T(n) = \vartheta(f(n))$
∈>0, and f(n) satisfies the regularity condition af(n/b) ≤ cf(n) for c<1 and all sufficiently large n.

Peak Finding Problem (1D)

An element A[i] is a peak if it is not smaller than all its neighbor(s)
if i!=1,n: A[i]≥A[i-1] and A[i]≥A[i+1]
if i=1: A[1] ≥ A[2]
if i=n: A[n] ≥ A[n-1]

Algorithm 1: Scan from left to right and compare A[i] with its neighbours, exit when peak found. T(n)=Θ(n)

Algorithm 2: Start from the middle element and compare with neighbours.
If A[n/2-1]>A[n/2]: search for a peak among A[1]… A[n/2-1]
If A[n/2]<A[n/2+1]: search for a peak among A[n/2+1]… A[n]
Else: A[n/2] is a peak!
Worst case: T(n) = T(n/2) + O(1) = O(log n)

---> time for comparing A[n/2] w 2 neighbours

Peak Finding (2D)

Algorithm 1: Brute Force. Worst case: $O(n^2)$
XX cannot find 1D peak on 1 col & 1D peak on the same row cause 2D peak may not exist on row i !!

Algorithm 2: Not efficient yet
(a) For each col j, find global max
(b) Apply 1D peak finder to find a peak among the max array
1D algorithm would solve max array in O(lg n) time but θ(n^2) time to compute max array

Algorithm 3:
Start from middle column j=m/2 n rows, m cols
Find global max on col j at (i,j)
Compare (i,j-1),(i,j)&(i,j+1). Pick Left columns if (i,j-1)>(i,j), vice versa. IF (i,j)>other2: ISSSA peak!!
Recursively start from middle again.
T(n,m) = T(N/2,m) + θ(m) = m log n [if find max along column]

Heaps:

Priority queue: implementing a set S of elements supporting the following operations: k(x) = key of element x
max(S): return element w largest key
insert(S,x): insert element x into S
extract_max(S): return element from S w largest key and remove it
increase_key(S,x,k'): increase val of x's key to new value, k'

Operation	Order they arrive	↓ing order	Heap
Max(S)	Θ(n)	Θ(1)	Θ(1)
Insert(S,x)	Θ(1)	Θ(n)	Θ(log n)
Extract_max(S)	Θ(n)	Θ(n)	Θ(log n)
Increase_key	Θ(n)	Θ(n)	Θ(log n)