Midterm:

One letter-size cheat-sheet is allowed (both sides).

Heaps, Priority Queues and Heap Sort:

Priority queue: a data structure that allows holding data with priority in extraction of data. Usage example: operating system task manager.
Queue: A simple priority queue: regular queue (FIFO). Insertion is done in constant time, but extract-max (or min) is $\Theta(n)$.
Skip-list: a multilayer like structure that allows insertion in logarithmic time. It allows binary search, therefore logarithmic.

Priority queue operations:

- Initialize
- Insert(key)
- Remove max

Unsorted array or linked list: insertion - $O(1)$, remove max - $O(n)$, delete - $O(1)$, average - $O(n)$.
Sorted array or list: insertion – $O(n)$, remove max – $O(1)$, delete – $O(1)$, average – $O(n)$
Heap: all operations are $O(\lg n)$

Heap

A binary tree storing keys with:

- Partial order: parent > children – you don't know the relation between the children (as opposed to a full order binary tree).
- Left-filled levels: all levels but last are full, last level is left filled.

Let n be the number of nodes, and $H(n) = \lg n$ is the height ⇒ $\boxed{2^{H(n)-1} \leq n \leq 2^{H(n)}}$
We use a simple <u>linear array</u> for representing the heap. The following is satisfied:

- Left_child(i) = 2i
- Right_child(i) = 2i + 1
- Parent(j) = j div 2

Since we're using an array, we want it to be left-filled – so each new node will be added to the end of the array (up to swapping between nodes).
<u>Remove max</u>:

- You remove the maximum, and take the last element and put it in the top of the tree
- Now to valid the heap the new top need to seep down, switching places every level with the highest of the two children (heapify down).
- Running time: $\lg n$

<u>Insert</u>:

- Put the new element at the end
- "Push" it up (heapify up).
- Running time: $\lg n$ – the maximum number of levels the heapify up operation will take.

The heapify algorithm:

If a node i is violating the heap condition, checking it is constant – access to $i, 2i, 2i+1$ is constant. The operations:

- Set the index for the largest of the elements in indices $i, 2i, 2i+1$.
- Swap if needed.

This process will have to be done at most $\lg n$, since a swap might violate the heap condition for the sub-tree from which a new parent is taken.

This algorithm can be used for implementation of all operations needed for the priority queue:

Extract max: remove the max, put the last element in the place of the first and heapify it down. Running time: $\lg n$.

Build a heap from an unsorted array:

The build algorithm starts from $n/2$ down to 1 since the node at $n/2$ is the first one that has a child, and may violate the heap condition.

Therefore from that node down to the first (1), you check the heap condition and heapify if needed. Running time: it's **NOT** $n \lg n$, as it has a more efficient amortized running time.

We have: $\frac{n}{2}$ at the first level, $\frac{n}{4}$ at the second level and so on – and each such node can go down at most that many level as the level it's at. When we sum it all up we get:

- Level 0 (from the lowest): $\frac{n}{2}$ lowest nodes are not touched – won't go down any level. Total: 0
- Level 1: $\frac{n}{4}$ nodes, each of them go down at most 2 levels. Total: $2 \times \frac{n}{2^2}$
- Level 2: $\frac{n}{8}$ nodes, each of them go down at most 3 levels. Total: $3 \times \frac{n}{2^3}$
- Level 3: $\frac{n}{16}$ nodes, each of them go down at most 4 levels. Total: $4 \times \frac{n}{2^4}$

And so on. The total:

$$\frac{n+1}{2} \cdot 1 + \frac{n+1}{4} \cdot 2 + \frac{n+1}{8} \cdot 3 + \cdots + \frac{n+1}{2^{\lg(n+1)}} \cdot \lg(n+1) = (n+1) \sum_{k=1}^{\lg(n+1)} \frac{i}{2^i} = O(n)$$

And the sum is $= O(1)$ (proof by induction). Therefore the total cost of all heapify operations is: $\boxed{O(n)}$

So the total running time for building a heap from an unsorted array is linear.

Create a sorted array:

First, build a heap in $O(n)$. Next:

For all elements from n down to 2 you switch the top (maximum) with the last element, consider the new last element (the max) outside the heap, and heapify down the new element in the root until it's in place.

After every iteration the size of the heap shrinks by 1, and the tail is sorted.

This phase's running time is $O(n \lg n)$.

Advantage of this algorithm is that like quick sort it's **in place**.

Selection Problems:

Given n numbers, a selection problem is finding some index out of a sorted sequence of that n numbers. For instance, finding maximum or minimum are selection problems. Each of these two problems in an unsorted array are **linear**.

i^{th} order statistic problem:

Finding $x \in \{a_1, \dots, a_n\}$ that satisfies: $i-1$ of the elements in the sorted sequence $< a'_1 \leq \cdots \leq a'_n >$ are left of x and $n-i$ are right to it. From this definition:

- Max: nth o.s. - $\Theta(n)$
- Min: 1^{st} o.s. - $\Theta(n)$
- Median: $\frac{n}{2}$ o.s. - ?

Formal definition:

Input: an array A of numbers of size n and a number i.

Output: the element x in A that is larger than exactly $i-1$ other elements in A.

Worst case is: $O(n \lg n)$ – with the trivial solution of sorting the array and find the desired o.s.

The **partition** algorithm: given p, q, r, q is the pivot with p elements $\leq$ of it, and the other are $>$ than it. If $i = q$, we got our order statistic. If that's not the case, there's a recursive call:

- $i < q \Rightarrow$ the i^{th} o.s. is in the set **left** to q, so that means we need to find the i^{th} o.s. among the first $p - q + 1$ elements.
- $i > q \Rightarrow$ the i^{th} o.s. is in the set **right** to q, so that means we need to find the i^{th} o.s. among the

So in the worst case: $T(n) = T(\max\{q, n-q\}) + \Theta(n)$

Where $\Theta(n)$ is for the partition, and the maximum taken is the worst case – that means it will be on some end of the current set of numbers – either 1 or $n-1$.

Thus the recurrence is: $T(n) = T(n-1) + \Theta(n) = O(n^2)$ – worse than sorting.

If we partition in the middle: $T(n) = T\left(\frac{n}{2}\right) + \Theta(n) = \cdots = O(1) + n\sum_{i=1}^{\lg n} \frac{1}{2^i} = \Theta(n)$. the same goes for any log base bigger than 1 (such as when taking $\frac{99}{100}n$).

So:

- Lucky: $T(n) = T\left(\frac{9n}{10}\right) + \Theta(n) = \Theta(n)$
- Unlucky: $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

Assuming **partition** splits A into 2 sizes $k, n-k-1$:

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1} T(\max\{k, n-k-1\}) + \Theta(n) \leq \frac{2}{n}\sum_{k=\frac{n}{2}}^{n-1} T(k) + \Theta(n)$$

Plugging in the assumption: $T(k) \leq c \cdot k$, we get to the sum above being bounded from above by $c \cdot n \Rightarrow$

The average input will end up **linear** with high probability (or simply randomize the partition).

A good partition is that a **portion** of the whole goes to each side of the partition: αn to one side and $(1-\alpha)n$ to the other. If one of the sides gets some constant number of elements, that will harm the running time.

Finding the ith order statistic algorithm:

$Select(A, p, q, i)$ algorithm:

- Divide A to $\frac{n}{5}$ groups of size 5.
- Find the median of each group of 5 by brute force, and store them in a set A' of size $\frac{n}{5}$.
- Use $select\left(A', 1, \frac{n}{5}, \frac{n}{10}\right)$ to find the median x of $\frac{n}{5}$ medians.
- Partition the n elements around x.for $k = q - p + 1$:
  - If $k = i$: return x

- If $i < k$: apply $select(A, p, k = 1, i)$
- If $i > k$: apply $select(A, k, q, i - k)$

The small sort operations in the second step take in total $c \cdot \frac{n}{5}$ for c being the constant time for sorting a set of 5 elements.

Applying $x = select\left(A', 1, \frac{n}{5}, \frac{n}{10}\right)$, we get that median for A'. That means: out of the columns (of size 5) to the left of the column that holds x, 3 of each column are smaller than x. That means: $3 \cdot \frac{n}{10}$ elements are smaller, and by the same logic on the right side $3 \cdot \frac{n}{10}$ are larger.

The above guaranties that: to the **left** you end up with at least $\frac{3n}{10}$ elements that are **smaller** than x, and at least $\frac{3n}{10}$ will go to the **right** and are **larger** than x. We have $\geq \frac{n}{4}$ elements smaller than x, meaning:

$\boxed{T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) + \Theta(n)}$ - $\frac{n}{5}$ for finding the median of $\frac{n}{5}$ medians, $\frac{3n}{4}$ the complexity of the recursion step (as we eliminated $\frac{n}{4}$ as smaller than the median we're looking). Note that we got recursive calls for:

- Total less than n
- Proportional parts of n

We conclude that: $T(n) = cn \Rightarrow \boxed{T(n) = O(n)}$ (proof in the lecture).

Using any 3 will **not** give the same results.

Using this in quick-sort gives us a worst case of $n \lg n$ (!).