**Some useful slides from cs240**

### Order Notation Summary

- **O-notation:** \( f(n) \in O(g(n)) \) if there exist constants \( c > 0 \) and \( n_0 > 0 \) such that \( |f(n)| \leq c |g(n)| \) for all \( n \geq n_0 \).

- **Ω-notation:** \( f(n) \in Ω(g(n)) \) if there exist constants \( c > 0 \) and \( n_0 > 0 \) such that \( c |g(n)| \leq |f(n)| \) for all \( n \geq n_0 \).

- **Θ-notation:** \( f(n) \in Θ(g(n)) \) if there exist constants \( c_1, c_2 > 0 \) and \( n_0 > 0 \) such that \( c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)| \) for all \( n \geq n_0 \).

- **ω-notation:** \( f(n) \in ω(g(n)) \) if for all constants \( c > 0 \), there exists a constant \( n_0 > 0 \) such that \( |f(n)| < c |g(n)| \) for all \( n \geq n_0 \).

- **o-notation:** \( f(n) \in o(g(n)) \) if for all constants \( c > 0 \), there exists a constant \( n_0 > 0 \) such that \( |f(n)| < c |g(n)| \) for all \( n \geq n_0 \).

### Useful Sums

**Arithmetic sequence:**
\[
\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \in Θ(n^2) \quad \text{if } d \neq 0.
\]

**Geometric sequence:**
\[
\sum_{i=0}^{n-1} r^i = \frac{r^n - 1}{r - 1} \in Θ(n^{r-1}) \quad \text{if } r > 1
\]
\[
\sum_{i=0}^{n-1} na^i = \frac{na^n - 1}{1 - a} \in Θ(n) \quad \text{if } r = 1
\]
\[
\sum_{i=0}^{n-1} \frac{1}{1 - r^i} \in Θ(1) \quad \text{if } 0 < r < 1.
\]

**Harmonic sequence:**
\[
\sum_{i=0}^{n-1} \frac{1}{i} = H_n := \sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + o(1) \in Θ(\log n)
\]

### Techniques for Order Notation

Suppose that \( f(n) > 0 \) and \( g(n) > 0 \) for all \( n \geq n_0 \). Suppose that
\[
L = \lim_{n \to \infty} \frac{f(n)}{g(n)}
\]

Then
\[
\begin{align*}
\Theta(g(n)) & \quad \text{if } L = 0 \\
ob(g(n)) & \quad \text{if } 0 < L < \infty \\
ω(g(n)) & \quad \text{if } L = \infty.
\end{align*}
\]

The required limit can often be computed using *l'Hôpital's rule*. Note that this result gives *sufficient* (but not necessary) conditions for the stated conclusions to hold.

### Useful Math Facts

**Logarithms:**
\[
c = \log_a(b) \text{ means } b^c = a. \quad \text{E.g. } n = 2^{\log n}. \text{ Requires } b > 1.
\]

\[
\log(a) \text{ (in this course) means } \log_2(a)
\]

\[
\log(a \cdot c) = \log(a) + \log(c), \quad \log(a^x) = c \log(a)
\]

\[
\log_a(x) = \frac{\log_b(x)}{\log_b(a)} = \frac{\log_c(x)}{\log_c(a)}
\]

\[
\ln(x) = \text{natural log} = \log_e(x)
\]

\[
\alpha \log x + (1 - \alpha) \log y \leq \log(\alpha x + (1 - \alpha) y) \quad \text{for } 0 \leq \alpha \leq 1
\]

**Factorial:**
\[
n! := n(n-1)(n-2) \cdots 2 \cdot 1 = \# \text{ ways to permute } n \text{ elements}
\]

\[
\log(n!) = \log n + \log(n-1) + \cdots + \log 2 + \log 1 \in Θ(n \log n)
\]

**Probability and moments:**
\[
E[aX] = aE[X], \quad E[X + Y] = E[X] + E[Y] \quad \text{(linearity of expectation)}
\]

### Some Recurrence Relations

<table>
<thead>
<tr>
<th>Recursion</th>
<th>resolves to</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) = T(n/2) + Θ(1) )</td>
<td>( T(n) \in Θ(\log n) )</td>
<td>Binary search</td>
</tr>
<tr>
<td>( T(n) = 2 T(n/2) + Θ(n) )</td>
<td>( T(n) \in Θ(n \log n) )</td>
<td>Mergesort</td>
</tr>
<tr>
<td>( T(n) = 2 T(n/2) + Θ(\log n) )</td>
<td>( T(n) \in Θ(n) )</td>
<td>Heapify (→ later)</td>
</tr>
<tr>
<td>( T(n) = T(\lfloor n/2 \rfloor) + Θ(\log n) )</td>
<td>( T(n) \in Θ(\sqrt{n}) )</td>
<td>for some ( 0 &lt; c &lt; 1 )</td>
</tr>
<tr>
<td>( T(n) = T(\lfloor n/4 \rfloor) + Θ(1) )</td>
<td>( T(n) \in Θ(\sqrt{n}) )</td>
<td>Selection (→ later)</td>
</tr>
<tr>
<td>( T(n) = T(\sqrt{n}) + Θ(1) )</td>
<td>( T(n) \in Θ(\log \log n) )</td>
<td>Interpolation Search (→ later)</td>
</tr>
</tbody>
</table>

Once you know the result, it is (usually) easy to prove by induction.

Many more recursions, and some methods to find the result, in cs341.
HeapSort

Idea: PQ-Sort with heaps.
But: Use same input-array A for storing heap.

```
HeapSort(A, n)
1. // heapify
2. n ← A.size()
3. for i ← 0 to n − 1 do
4.   fix-down(A, i)
5. // repeatedly find maximum
6. while n > 1 do
7.   if k = A[root()] return ℓ
8.   else return "not found, but would be between ℓ − 1 and ℓ"

The for-loop takes Θ(n) time and the while-loop takes O(n log n) time.
```

Efficient In-Place partition (Hoare)

Idea: Keep swapping the outer-most wrongly-positioned pairs.

```
partition(A, p)
A: array of size n, p: integer s.t. 0 ≤ p < n
1. swap(A[p − 1], A[p])
2. i ← −1, j ← n + 1, v ← A[p - 1]
3. loop
4. do i ← i + 1 while i < j and A[i] < v
5. do j ← j − 1 while j > 0 and A[j] > v
6. if i ≥ j then break (goto 9)
7. else swap(A[i], A[j])
8. end loop
9. swap(A[n − 1], A[i])
10. return i
```

Running time: Θ(n).

Count Sort Pseudocode

```
count-sort(A, d)
A: array of size n, contains numbers with digits in {0, . . . , R − 1}
d: index of digit by which we wish to sort
1. count ← array of size R, filled with zeros
2. for i ← 0 to n − 1 do
3.   increment count[dth digit of A[i]]
4. // find left boundary for each kind
5. idx ← array of size R, idx[0] = 0
6. for i ← 1 to R − 1 do
7.   idx[i] ← idx[i − 1] + count[i − 1]
8. // move to new array in sorted order, then copy back
9. aux ← array of size n
10. for i ← 0 to n − 1 do
11.   aux[idx[dth digit of A[i]]] ← A[i]
12. increment idx[dth digit of A[i]]
13. return aux
```

Binary Search

Ordered array
insert, delete: Θ(n)
search: Θ(log n)

```
Binary-search(A, n, k)
A: Array of size n, k: key
1. ℓ ← 0
2. r ← n − 1
3. while (ℓ < r)
4.   m ← \frac{ℓ + r}{2}
5.   if (A[m] < k) ℓ = m + 1
6.   else if (k < A[m]) r = m − 1
7.   else return m
8. if (k = A[m]) return ℓ
9. else return "not found, but would be between ℓ − 1 and ℓ"
```