## Useful Math Facts

### Logarithms:
- $c = \log_b(a)$ means $b^c = a$. E.g. $2^{2\log_2 n} = n^2$.
- $\log(a)$ (in this course) means $\log_2(a)$.
- $\log(a \cdot c) = \log(a) + \log(c)$.
- $\log(a^2) = 2\log(a) = \log(a^a)$.
- $\ln(x)$ is natural log $\log_e(x)$.
- Convexity: $\alpha \log x + (1-\alpha) \log y \leq \log(\alpha x + (1-\alpha)y)$ for $0 \leq \alpha \leq 1$.

### Factorial:
- $n! := (n-1)! \cdot n$.
- $\log(n!) = \log(n) + \log(n-1) + \cdots + \log 1$.

### Probability and moments:
- $E[X] = a + \sum_{i=0}^{n-1} i$ (Arithmetic sequence).

### Some Recurrence Relations

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

### Efficient In-Place partition (Hoare)

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

### 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 ← parent(last(n)) downto 0 do
   4. fix-down(A, n, i)
 5. // repeatedly find maximum
 6. while n > 1
 7. // do deleteMax
 8. swap items at A[root()] and A[last(n)]
 9. decrease n
 10. fix-down(A, n, root())
```

The for-loop takes $\Theta(n)$ time and the while-loop takes $O(n \log n)$ time.
Count Sort Pseudocode

key-indexed-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
// count how many of each kind there are
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]]
// find left boundary for each kind
4. idx ← array of size R, idx[0] = 0
5. for i ← 1 to R − 1 do
6. idx[i] ← idx[i − 1] + count[i − 1]
// move to new array in sorted order, then copy back
7. aux ← array of size n
8. for i ← 0 to n − 1 do
10. increment idx[dth digit of A[i]]
11. A ← copy(aux)

Complexity of open addressing strategies

For any open addressing scheme, we must have α < 1 (why?).
Cuckoo hashing requires α < 1/2.

Avg.-case costs:

<table>
<thead>
<tr>
<th></th>
<th>search (unsuccessful)</th>
<th>search (successful)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Probing</td>
<td>( \frac{1}{(1-\alpha)^2} )</td>
<td>( \frac{1}{1-\alpha} )</td>
</tr>
<tr>
<td>Double Hashing</td>
<td>( \frac{1}{1-\alpha} )</td>
<td>( \frac{1}{1-\alpha} )</td>
</tr>
<tr>
<td>Cuckoo Hashing</td>
<td>( \frac{1}{1-2\alpha^2} )</td>
<td>( \frac{1}{1-\alpha} )</td>
</tr>
</tbody>
</table>

Summary: All operations have \( O(1) \) average-case run-time if the hash-function is uniform and \( \alpha \) is kept sufficiently small. But worst-case run-time is (usually) \( \Theta(n) \).

String Matching Conclusion

- Brute-Force
- Karp-Rabin
- Boyer-Moore
- DFA
- Knuth-Morris-Pratt
- Suffix Tree
- Suffix Array

<table>
<thead>
<tr>
<th>Operation</th>
<th>Brute-Force</th>
<th>Karp-Rabin</th>
<th>Boyer-Moore</th>
<th>DFA</th>
<th>Knuth-Morris-Pratt</th>
<th>Suffix Tree</th>
<th>Suffix Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preproc.</td>
<td>( O(m) )</td>
<td>( O(m+</td>
<td>Σ</td>
<td>) )</td>
<td>( O(m</td>
<td>Σ</td>
<td>) )</td>
</tr>
<tr>
<td>Search</td>
<td>( O(nm) )</td>
<td>( O(m+n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(m) )</td>
<td>( O(m\log n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Extra space</td>
<td>( O(1) )</td>
<td>( O(m+</td>
<td>Σ</td>
<td>) )</td>
<td>( O(m</td>
<td>Σ</td>
<td>) )</td>
</tr>
</tbody>
</table>

- Code very similar to binary search, but compare at interpolated index
- Need a few extra tests to avoid crash due to \( A[i] = A[r] \)

Interpolation Search

\[ \text{interpolation-search}(A, n, k) \]
A: Array of size n, k: key
1. \( \ell \leftarrow 0 \)
2. \( r \leftarrow n - 1 \)
3. while \( (\ell < r) \) && \( (A[\ell] = A[r]) \) && \( (k \geq A[\ell]) \) && \( (k \leq A[r]) \)
4. \( m \leftarrow \lfloor \frac{\ell + r}{2} \rfloor \)
5. if \( (A[m] < k) \)
6. \( \ell \leftarrow m + 1 \)
7. else \( r \leftarrow m - 1 \)
8. if \( (k = A[\ell]) \) return \( \ell \)
9. else return "not found, but would be between \( \ell - 1 \) and \( \ell + 1" \)

Range query data structures summary

- Quadtrees
  - simple (also for dynamic set of points)
  - work well only if points evenly distributed
  - wastes space for higher dimensions
- kd-trees
  - linear space
  - query-time \( O(\sqrt{n}) \)
  - inserts/deletes destroy balance
  - care needed for duplicate coordinates
- range trees
  - fastest range search \( O(\log^2 n) \)
  - wastes some space
  - insert and delete more complicated

Convention: Points on split lines belong to right/top side.

Compression summary

- Huffman run-length encoding
- Lempel-Ziv-Welch
- bzip2 (uses Burrows-Wheeler)

<table>
<thead>
<tr>
<th>Method</th>
<th>Huffman</th>
<th>Run-length Encoding</th>
<th>Lempel-Ziv-Welch</th>
<th>bzip2 (uses Burrows-Wheeler)</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable-length</td>
<td>variable-length</td>
<td>fixed-length</td>
<td>multi-step</td>
<td></td>
</tr>
<tr>
<td>single-character</td>
<td>multi-character</td>
<td>multi-character</td>
<td>multi-step</td>
<td></td>
</tr>
<tr>
<td>2-pass, must send dictionary</td>
<td>1-pass</td>
<td>1-pass</td>
<td>not streamable</td>
<td></td>
</tr>
<tr>
<td>60% compression on English text</td>
<td>bad on text</td>
<td>45% compression on English text</td>
<td>70% compression on English text</td>
<td></td>
</tr>
<tr>
<td>optimal 01-prefix code</td>
<td>good on long runs (e.g., pictures)</td>
<td>good on English text</td>
<td>better on English text</td>
<td></td>
</tr>
<tr>
<td>requires uneven frequencies</td>
<td>requires runs</td>
<td>requires repeated substrings</td>
<td>requires repeated substrings</td>
<td></td>
</tr>
<tr>
<td>rarely used directly</td>
<td>rarely used directly</td>
<td>frequently used</td>
<td>used but slow</td>
<td></td>
</tr>
<tr>
<td>part of pkzip,</td>
<td>fax machines, old picture-formats</td>
<td>GZIP, some variants of PDF, compress</td>
<td>bzip2 and variants</td>
<td></td>
</tr>
<tr>
<td>JPEG, MP3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>