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>$T(n) \in \Theta(\log n)$</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>$T(n) = 2T(n/2) + \Theta(\log n)$</td>
<td>$T(n) \in \Theta(n)$</td>
<td>Heapify ($\rightarrow$ 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 ($\rightarrow$ later)</td>
</tr>
<tr>
<td>$T(n) = 2T(n/4) + \Theta(1)$</td>
<td>$T(n) \in \Theta(\sqrt{n})$</td>
<td>Range Search ($\rightarrow$ 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 ($\rightarrow$ later)</td>
</tr>
</tbody>
</table>

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 $n \geq n_0$.

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

$\Theta$-notation: $f(n) \in \Theta(g(n))$ 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 $n \geq n_0$.

$o$-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 $n \geq n_0$.

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

Useful Sums

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

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

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

**A few more:**
\[
\sum_{i=1}^{n} \frac{1}{i^2} = \sum_{i=1}^{n} \frac{1}{i^2} = \frac{\pi^2}{6} \in \Theta(1)
\]
\[
\sum_{i=1}^{n} i^k = \sum_{i=1}^{n} i^k \in \Theta(n^{k+1}) \quad \text{for} \quad k \geq 0
\]

Useful Math Facts

**Logarithms:**
\[
c = \log_b(a) \quad \text{means} \quad b^c = a. \quad \text{E.g.} \quad n = 2^{\log n}.
\]
\[
\log(a) \quad \text{(in this course) means} \quad \log_2(a)
\]
\[
\log(a \cdot c) = \log(a) + \log(c), \quad \log(a^c) = c \log(a)
\]
\[
\log_b(a) = \log_{10}(a) \cdot \frac{1}{\log_{10}(b)}, \quad a^{\log_b c} = c^{\log_b a}
\]
\[
\ln(x) = \text{natural log} = \log_e(x), \quad \frac{d}{dx} \ln x = \frac{1}{x}
\]
\[
\text{concavity:} \quad \alpha \log x + (1-\alpha) \log y \leq \log(\alpha x + (1-\alpha)y) \quad \text{for} \quad 0 \leq \alpha \leq 1
\]

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

**Probability and moments:**
\[
E[aX] = aE[X], \quad E[X + Y] = E[X] + E[Y] \quad \text{(linearity of expectation)}
\]
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 $\{0, \ldots, R-1\}$
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[$d^{th}$ 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
9. aux[idx[$d^{th}$ digit of $A[i]]] ← A[i]
10. increment idx[$d^{th}$ digit of $A[i]]$
11. A ← copy(aux)
```

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. $0 \leq p < n$
1. swap(A[p-1], A[p])
2. $i$ ← $p-1$, $j$ ← $n-1$, $v$ ← $A[n-1]$;
3. loop
4. do $i$ ← $i + 1$ while $i < n$ and $A[i] < v$
5. do $j$ ← $j - 1$ while $j > 0$ and $A[j] > v$
6. if $i \geq j$ then break (goto 9)
7. else swap($A[i], A[j]$)
8. end loop
9. swap($A[p-1], A[i])$
10. return $i$
```

Running time: $\Theta(n)$.

Interpolation Search

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

```
Interpolation-search(A, n, k)
A: Array of size $n$, $k$: key
1. $\ell$ ← 0
2. $r$ ← $n - 1$
3. while $(\ell < r)$ && $(A[\ell] = A[r])$ && $(k \geq A[\ell])$ && $(k < A[r])$
4.   $m$ ← $\lfloor \frac{\ell + \lfloor \frac{A[r]-A[\ell]}{r-\ell} \rfloor \cdot (r-\ell) \rfloor$
5.   if $(A[m] < k)$ then $\ell$ ← $m + 1$
6.   else if $(A[m] = k)$ return $m$
7.   else $r$ ← $m - 1$
8. if $(k = A[\ell])$ return $\ell$
9. else return "not found, but would be between $\ell - 1$ and $\ell"
```
Complexity of open addressing strategies
For any open addressing scheme, we must have $\alpha < 1$ (why?). Cuckoo hashing requires $\alpha < 1/2$.

<table>
<thead>
<tr>
<th>Avg.-case costs:</th>
<th>search (unsuccessful)</th>
<th>insert</th>
<th>search (successful)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Probing</td>
<td>$\frac{1}{(1-\alpha)^2}$</td>
<td>$\frac{1}{(1-\alpha)^2}$</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>
<td>$\frac{1}{\alpha \log \left( \frac{1}{1-\alpha} \right)}$</td>
</tr>
<tr>
<td>Cuckoo Hashing</td>
<td>$\frac{1}{(1-2\alpha)^2}$ (worst-case)</td>
<td>$\frac{1}{\alpha}$ (worst-case)</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) $\Theta(n)$.

Knuth-Morris-Pratt Algorithm

**KMP**$(T,P)$
1. $F \leftarrow \text{failureArray}(P)$
2. $i \leftarrow 0$ // current character of $T$ to parse
3. $j \leftarrow 0$ // current state that we are in
4. while $i < n$ do
5. if $P[j] = T[i]$
6. if $j = m - 1$
7. return "found at guess $i - m + 1$"
8. else
9. $i \leftarrow i + 1$
10. $j \leftarrow j + 1$
11. // i.e. $P[j] \neq T[i]$
12. if $j > 0$
13. $j \leftarrow F[j - 1]$
14. else
15. $i \leftarrow i + 1$
16. return FAIL

Boyer-Moore Algorithm
Brute-force search with two changes:
- **Reverse-order searching:** Compare $P$ with a guess moving backwards
- **Bad character jumps:** When a mismatch occurs, then eliminate guesses where $P$ does not agree with this char of $T$
- In practice large parts of $T$ will not be looked at.

**BoyerMoore**$(T,P)$
1. $L \leftarrow$ last occurrence array computed from $P$
2. $i \leftarrow 0$ // current guess
3. while $i \leq n - m$
4. for $(j \leftarrow m - 1, j \geq 0, j--)$
5. if $T[i+j] \neq P[j]$ break
6. if $j = -1$ return "found at guess $i"
7. else $i \leftarrow i + \max\{1,j-L[T[i+j]]\}$
8. return FAIL

$L$ will be explained below.

Compression Summary

<table>
<thead>
<tr>
<th>Huffman</th>
<th>Run-length encoding</th>
<th>Lempel-Ziv-Welch</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable-length</td>
<td>variable-length</td>
<td>fixed-length</td>
</tr>
<tr>
<td>single-character</td>
<td>multi-character</td>
<td>multi-character</td>
</tr>
<tr>
<td>2-pass</td>
<td>1-pass</td>
<td>1-pass</td>
</tr>
<tr>
<td>60% compression</td>
<td>bad on text</td>
<td>45% compression on English text</td>
</tr>
<tr>
<td>on English text</td>
<td></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>
</tr>
<tr>
<td>must send dictionary</td>
<td>can be worse than ASCII</td>
<td>can be worse than ASCII</td>
</tr>
<tr>
<td>rarely used directly</td>
<td>rarely used directly</td>
<td>frequently used</td>
</tr>
<tr>
<td>part of pkzip, JPEG, MP3</td>
<td>fax machines, old picture-formats</td>
<td>GIF, some variants of PDF, Unix compress</td>
</tr>
</tbody>
</table>
Quad Tree and kd-tree convention

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

Karp-Rabin Fingerprint Algorithm – Conclusion

```
Karp-Rabin-RollingHash(T, P)
1.   hp ← h(P[0..m−1])
2.   p ← suitable prime number
3.   s ← 10^{m−1} mod p
4.   h_T ← h(T[0..m−1])
5.   for i ← 0 to n − m
6.       if i > 0 // compute hash-value for next guess
7.           h_T ← ((h_T − T[i] · s) · 10 + T[i+m]) mod p
8.       if h_T = hp
9.           if strcmp(T[i..i+m−1], P) = 0
10.          return “found at guess i”
11.         return "FAIL"
```

- Choose “table size” \( p \) at random to be huge prime
- Expected running time is \( O(m + n) \)
- \( \Theta(mn) \) worst-case, but this is (unbelievably) unlikely