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>$T(n) ∈ Θ(log n)$</td>
<td>Binary search</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + Θ(n)$</td>
<td>$T(n) ∈ Θ(n log n)$</td>
<td>Mergesort</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + Θ(log n)$</td>
<td>$T(n) ∈ Θ(n)$</td>
<td>Heapify (→ later)</td>
</tr>
<tr>
<td>$T(n) = T(cn) + Θ(n)$ for some $0 &lt; c &lt; 1$</td>
<td>$T(n) ∈ Θ(n)$</td>
<td>Selection (→ later)</td>
</tr>
<tr>
<td>$T(n) = 2T(n/4) + Θ(1)$</td>
<td>$T(n) ∈ Θ(√n)$</td>
<td>Range Search (→ later)</td>
</tr>
<tr>
<td>$T(n) = T(√n) + Θ(1)$</td>
<td>$T(n) ∈ Θ(log log n)$</td>
<td>Interpolation Search (→ later)</td>
</tr>
</tbody>
</table>

Order Notation Summary

$O$-notation: $f(n) ∈ O(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $|f(n)| ≤ cg(n)$ for all $n ≥ n_0$.

$Ω$-notation: $f(n) ∈ Ω(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $|f(n)| ≥ cg(n)$ for all $n ≥ n_0$.

$Θ$-notation: $f(n) ∈ Θ(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $c_1g(n) ≤ |f(n)| ≤ c_2g(n)$ for all $n ≥ n_0$.

$o$-notation: $f(n) ∈ o(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $|f(n)| < cg(n)$ for all $n ≥ n_0$.

$ω$-notation: $f(n) ∈ ω(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $c |g(n)| < |f(n)|$ for all $n ≥ n_0$.

Useful Sums

**Arithmetic sequence:**

$$\sum_{i=0}^{n-1} i = ??? \quad \sum_{i=0}^{n-1} (a + di) = na + \frac{d(n^2-1)}{2} ∈ Θ(n^2) \text{ if } d ≠ 0.$$  

**Geometric sequence:**

$$\sum_{i=0}^{n-1} a^i = ??? \quad \sum_{i=0}^{n-1} a^i r = \begin{cases} a^r - 1 & ∈ Θ(r^{n-1}) \text{ if } r > 1 \\ na & ∈ Θ(n) \text{ if } r = 1 \\ a^1 - r^1 & ∈ Θ(1) \text{ if } 0 < r < 1. \end{cases}$$

**Harmonic sequence:**

$$\sum_{i=0}^{n-1} \frac{1}{i} = ??? \quad H_n := \sum_{i=0}^{n-1} \frac{1}{i} = ln n + γ + o(1) ∈ Θ(log n)$$

A few more:

$$\sum_{i=0}^{n-1} \frac{1}{i^k} = ??? \quad \sum_{i=0}^{n-1} \frac{1}{i} ∈ Θ(1) \quad \sum_{i=0}^{n-1} i^k ∈ Θ(n^{k+1}) \quad \text{for } k ≥ 0$$

HeapSort

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

```java
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.   // delete the maximum
8.   swap items at A[root()] and A[last(n)])
9.   decrease n
10. fix-down(A, n, root())
```

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

Useful Math Facts

**Logarithms:**

- $c = log_b(a)$ means $b^c = a$. E.g. $n = 2^{log_2 n}$.
- $\log(a)$ (in this course) means $\log_{10}(a)$.
- $\log(a ∙ c) = \log(a) + \log(c)$, $\log(a^x) = x \log(a)$.
- $\log_a(b) = \frac{\log_c(b)}{\log_c(a)}$, $\log_b c = \frac{\log_a c}{\log_a b}$.
- $\ln(x) = \text{natural log} = \log_e(x)$, $\frac{d}{dx} \ln x = \frac{1}{x}$.
- concavity: $\alpha \log x + (1-\alpha) \log y ≤ \log(ax+(1-\alpha)y)$ for $0 ≤ \alpha ≤ 1$.

**Factorial:**

- $n! := n(n-1)(n-2)⋯2·1 = \#$ ways to permute n elements.
- $\log(n!) = \log n + \log(n-1) + ⋯ + \log 2 + \log 1 ∈ Θ(n log n)$

**Probability and moments:**


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[n-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 ≥ j then break (goto 9)
7. else swap(A[i], A[j])
8. end loop
9. swap(A[i - 1], A[i])
10. return i
```

Running time: $Θ(n)$. 

Abelard, Biniaz, Schost (SCS, UW)  CS240 – Module 00  Spring 2019 6 / 12
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[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 \( \alpha < 1/2 \).
Cuckoo hashing requires \( \alpha < 1/2 \).

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Avg.-case costs:</th>
<th>\text{search (unsuccessful)}</th>
<th>\text{insert}</th>
<th>\text{search (successful)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Probing</td>
<td>( 1 )</td>
<td>( \frac{1}{1 - \alpha} )</td>
<td>( \frac{1}{1 - \alpha} )</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}{1 - \alpha} )</td>
<td>( \frac{1}{1 - \alpha} )</td>
</tr>
<tr>
<td>Cuckoo Hashing</td>
<td>( \frac{1}{\text{worst-case}} )</td>
<td>( \frac{1}{\text{worst-case}} )</td>
<td>( \frac{1}{\text{worst-case}} )</td>
<td>( \frac{1}{\text{worst-case}} )</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

<table>
<thead>
<tr>
<th>Scheme</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>\Sigma</td>
<td>) )</td>
<td>( O(m) )</td>
<td>( O(n^2) )</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Search time</td>
<td>( O(nm) )</td>
<td>( O(nm) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(m) )</td>
<td>( O(n m \log n) )</td>
<td></td>
</tr>
<tr>
<td>Extra space</td>
<td>( O(1) )</td>
<td>( O(m +</td>
<td>\Sigma</td>
<td>) )</td>
<td>( O(m) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

- Our algorithms stopped once they have found one occurrence.
- Most of them can be adapted to find all occurrences within the same worst-case run-time.

Compression summary

<table>
<thead>
<tr>
<th>Scheme</th>
<th>variable-length</th>
<th>variable-length</th>
<th>fixed-length</th>
<th>multi-step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huffman</td>
<td>single-character</td>
<td>multi-character</td>
<td>multi-character</td>
<td>multi-step</td>
</tr>
<tr>
<td>Run-length</td>
<td>1-pass</td>
<td>1-pass</td>
<td>not streamable</td>
<td></td>
</tr>
<tr>
<td>encoding</td>
<td>60% compression</td>
<td>bad on text</td>
<td>45% compression</td>
<td>70% compression</td>
</tr>
<tr>
<td>Lempel-Ziv-Welch</td>
<td>optimal</td>
<td>good on long runs(e.g., pictures)</td>
<td>good on English text</td>
<td>better on English text</td>
</tr>
<tr>
<td></td>
<td>01-prefix_code</td>
<td>requires uneven frequencies</td>
<td>requires repeated substrings</td>
<td>requires repeated substrings</td>
</tr>
<tr>
<td></td>
<td>rarely used directly</td>
<td>rarely used directly</td>
<td>frequently used</td>
<td>used but slow</td>
</tr>
<tr>
<td></td>
<td>part of pkzip, JPEG, MP3</td>
<td>part of pkzip, JPEG, MP3</td>
<td>part of pkzip, JPEG, MP3</td>
<td>part of pkzip, JPEG, MP3</td>
</tr>
<tr>
<td></td>
<td>GZIP, some variants of PDF, compress</td>
<td>GZIP, some variants of PDF, compress</td>
<td>GZIP, some variants of PDF, compress</td>
<td>GZIP, some variants of PDF, compress</td>
</tr>
</tbody>
</table>

Interpolation Search

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

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 \( O(\sqrt{\pi + s}) \)
  - inserts/deletes destroy balance
  - care needed if not in general position
- Range trees
  - query-time \( O(\log^s n + s) \)
  - wastes some space
  - inserts/deletes destroy balance

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