Some useful slides from cs240E

Order Notation Summary

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

Ω-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 \( n \geq n_0 \).

Θ-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 \( f(n) > c g(n) \) for all \( n \geq n_0 \).

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 \( \log_2(a) \).
* \( \log(a \cdot c) = \log(a) + \log(c) \).
* \( \log(a^x) = x \log(a) \).
* \( \ln(x) \) = natural log = \( \log_e(x) \).
* Concavity: \( \log x + (1-\alpha) y \leq \log(c x + (1-\alpha) y) \) for \( 0 \leq \alpha \leq 1 \).

Factorial:
* \( n! := n(n-1)(n-2) \cdots 2 \cdot 1 \) \# ways to permute \( n \) elements
* \( \log(n!) \) is \( \log n + \log(n-1) + \cdots + \log 2 + \log 1 \in \Theta(n \log n) \).

Probability and moments:
* \( \text{E}([aX] = a \text{E}[X], \text{E}[X + Y] = \text{E}[X] + \text{E}[Y] \) (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 do
7.      // do deleteMax
8.      swap items at A[\text{root()}] and A[\text{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.

Useful Sums

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

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

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

**A few more:**
\[
\sum_{i=1}^{n} \frac{1}{i} \not\in \Theta(1)
\]
\[
\sum_{i=1}^{n} k^i \not\in \Theta(n^{k+1}) \text{ for } k \geq 0
\]

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>Heapsify ((\rightarrow \text{ 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 \text{ 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 \text{ 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 \text{ 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.

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[n-1], A[p]]
2.  i ← 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[n-1], A[i]]
10.  return i
```

Running time: \( \Theta(n) \).
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 \leftarrow 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 \leftarrow 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 \leftarrow 0 to n - 1 do
10. increment idx[dth digit of A[i]]
11. A \leftarrow copy(aux)

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</th>
<th>insert</th>
<th>search</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unsuccessful)</td>
<td>( \frac{1}{1 - \alpha} )</td>
<td>( \frac{1}{(1 - \alpha)^2} )</td>
<td>( \frac{1}{1 - \alpha} )</td>
</tr>
<tr>
<td>Linear Probing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double Hashing</td>
<td>( \frac{1}{1 - \alpha} )</td>
<td>( \frac{1}{1 - \alpha} )</td>
<td>( \frac{1}{\log(1 - \alpha)} )</td>
</tr>
<tr>
<td>Cuckoo Hashing</td>
<td></td>
<td></td>
<td>( \frac{1}{1 - 2\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) \( O(n) \).

String Matching Conclusion

<table>
<thead>
<tr>
<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 1</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>\Sigma</td>
</tr>
<tr>
<td>Search time</td>
<td>( O(nm) )</td>
<td>( O(n + \text{expected}) )</td>
<td>( O(n) ) or better</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(m \log n) )</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>\Sigma</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.

1 studied only in the enriched section

Interpolation Search

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

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
  - variable-length encoding
- Run-length encoding
  - fixed-length
- Lempel-Ziv-Welch
  - multi-step
- bzip2 (uses Burrows-Wheeler)
  - multi-step

| Variable-length | Multi-character
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Good on runs (e.g., pictures)</td>
<td>Good on English text</td>
</tr>
<tr>
<td>Requires uneven frequencies</td>
<td>Requires repeated substrings</td>
</tr>
<tr>
<td>Rarely used directly</td>
<td>Frequently used</td>
</tr>
<tr>
<td>Part of pkzip, JPEG, MP3</td>
<td>GIF, some variants of PDF, compress</td>
</tr>
</tbody>
</table>
Huffman’s Algorithm: Building the best trie

For a given source text $S$, how to determine the “best” trie that minimizes the length of $C^T$?

- Determine frequency of each character $c \in \Sigma$ in $S$
- For each $c \in \Sigma$, create a trie (height-0 trie holding $c$).
- Our tries have a weight: sum of frequencies of all letters in trie. Initially, these are just the character frequencies.
- Find the two tries with the minimum weight.
- Merge these tries with new interior node; new weight is the sum.
- Repeat last two steps until there is only one trie left

What data structure should we store the tries in to make this efficient?

Prefix-free Encoding for Positive Integers

Use Elias gamma coding to encode $k$:
- $\lfloor \log k \rfloor$ copies of 0, followed by
- binary representation of $k$ (always starts with 1)

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\lfloor \log k \rfloor$</th>
<th>$k$ in binary</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>11</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>100</td>
<td>1000100</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>101</td>
<td>001011</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>110</td>
<td>001110</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

RLE Encoding pseudocode

```
RLE-encoding(S[0..n-1])
S: string
1. Initialize output string C ← S[0]
2. i ← 0 // index of parsing S
3. while i < n do
4.   k ← 1 // length of run
5.   while (i + k < n and S[i + k] = S[i]) do k ← k + 1
6.   j ← i + k
7.   // compute and append Elias gamma code
8.   K ← empty string
9.   while k > 1
10.    C.append(0) // index of parsing S
11.    k ← k \mod 2
12.    K.append(1) // K is binary encoding of k
13. C.append(K)
14. return C
```

RLE Decoding pseudocode

```
RLE-decoding(C)
C: stream of bits
1. Initialize output string S
2. b ← C.pop() // bit-value for the current run
3. repeat
4.   ℓ ← 0 // length of base-2 number − 1
5.   while C.pop() = 0 do ℓ ← ℓ + 1
6.   k ← 1 // base-2 number converted
7.   s ← (ℓ + 1) \mod 2 + C.pop() // bit-value for the current run
8.   s ← s + 2 \cdot C.pop() // special situation!
9.   k ← k + 1 − 2 \cdot s
10. until C has no more bits left
11. return S
```

LZW encoding pseudocode

```
LZW-encoding(S)
S: stream of characters
1. Initialize dictionary D with ASCII in a trie
2. id ← 128
3. while there is input in S do
4.   v ← root of trie D
5.   K ← S.peek()
6.   while (v has a child K labelled K)
7.     v ← D[S.pop()]
8.   if there is no more input in S break. (goto 10)
9.   S.pop()
10. output codenumber stored at v
11. if there is more input in S
12. create child of v labelled K with codenumber id
13. id ← id + 1
```

LZW decoding pseudocode

```
LZW-decoding(C)
C: stream of integers
1. D ← dictionary that maps {0, ..., 127} to ASCII
2. id ← 128
3. S ← empty string
4. code ← C.pop(); s ← D(code); S.append(s)
5. while there are more codes in C do
6.   \textit{special situation!}
7.   if code < id
8.     id ← id + 1
9.   s ← D(code)
10. else if code = id
11. then FAIL // Encoding was invalid
12. S.append(s)
13. D.insert(id, S.pop(); \textit{special situation!})
14. id ← id + 1
15. return S
```
Computing the Failure Array

\[
\text{failureArray}(P)
\]
\[P: \text{String of length } m \text{ (pattern)}\]
1. \[F[0] \leftarrow 0\]
2. \[i \leftarrow 1\]
3. \[j \leftarrow 0\]
4. \[\textbf{while } i < m \textbf{ do}\]
5. \[\textbf{if } P[i] = P[j]\]
6. \[j \leftarrow j + 1\]
7. \[F[i] \leftarrow j\]
8. \[i \leftarrow i + 1\]
9. \[\textbf{else if } j > 0\]
10. \[j \leftarrow F[j - 1]\]
11. \[\textbf{else}\]
12. \[F[i] \leftarrow 0\]
13. \[i \leftarrow i + 1\]

Correctness-idea: \[F[\cdot]\] is defined via pattern matching of \(P[1..j]\) in \(P\). So KMP uses itself! Already-built parts of \(F[\cdot]\) are used to expand it.

Abéard, Biniaz, Schost (SCS, UW) CS240 – Module 9 Spring 2019 22 / 35

Suffix Array Building

\[\text{Text } T: \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}\]
\[B_j: \text{Index of } T[i..n-1] \text{ in } A_j: \]

<table>
<thead>
<tr>
<th>(i)</th>
<th>suffix</th>
<th>(A_0) suffix</th>
<th>(A_1) suffix</th>
<th>(A_2) suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bananaban</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ananaban</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>nanaban</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>anaban</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>naban</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>aban</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ban</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>an</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Count-sort by first letter
- For all groups, determine partners and their indices
- Sort by partner-indices and repeat.