COMP 252: Summary

Anna Brandenberger

April 28, 2018

This is a condensed scribing of the lectures given by Luc Devroye in the Winter semester of 2018 for the Honours Data Structures and Algorithms class (COMP 252, McGill University). Subjects covered: Divide-and-Conquer algorithm examples, Dynamic Programming, various Abstract Data Types and implementations.

Definitions & Theorems

Introduction

Definition 1. An **algorithm** is a finite set of instructions which takes in a finite number of inputs, performs a task exactly, produces a finite output and exits. (c.f. Church-Turing Thesis.)

Definition 2. We examine two **cost models**:

1. *Uniform cost model:* every operation takes 1 time unit.
2. *Bitwise (logarithmic) cost model:* every *bit* operation takes 1 time unit (i.e. basic operations like reading, shifting, etc.).

Complexity

Definition 3. An **oracle** is a black-box type device capable of solving some computational problem in a single time unit: takes in input(s) $x_1, \cdots, x_n$ and provides output(s) $y_1, \cdots, y_k$.

Definition 4. O **- Big O**: upper bound. We say that $a_n = O(b_n)$ if $\exists c > 0$, $n_0 > 0$ such that $a_n \le cb_n$, $\forall n \ge n_0$.

Definition 5. Ω **- Big Omega**: lower bound. We say that $a_n = \Omega(b_n)$ if $\exists c > 0, n_0 > 0$ such that $a_n \ge cb_n$, $\forall n \ge n_0$.

Lower Bounds chapter: if the lower bound for solving a problem with input of size n is f_n, it implies that $\forall$ algorithms (most efficient), $\exists$ an input $x_1, \cdots, x_n$ (worst-case) s.t. T(algorithm, $x_1, \cdots, x_n) \ge f_n$.

Definition 6. Θ **- Big Theta**: precise asymptotic behaviour. We say that $a_n = \Theta(b_n)$ if $a_n = O(b_n) = \Omega(b_n)$.

Recurrences

Theorem 7 (Master Theorem)**.** *Given a recurrence $T_n = aT_{\lfloor n/b \rfloor} + f(n)$,*

(a) $n^{\log_b a}/f(n) > n^{\epsilon} \;\; \forall n \ge n_0 \implies T_n = O(n^{\log_b a})$. [1]

(b) $f(n)/n^{\log_b a)} > n^{\epsilon} \;\; \forall n \ge n_0 \implies T_n = O(f(n))$. [2]

(c) $f(n) = \Theta(n^{\log_b a}) \implies T_n = \Theta(n^{\log_b a} \cdot \log_b n)$.

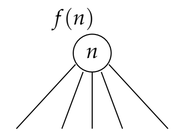
[1] If $n^{\log_b a}/f(n) = \Theta(n^c) \;\; \forall n \ge n_0$, then $T_n = \Theta(n^{\log_b a})$ instead.

[2] This case holds under the technical assumption

$$\lim_{n\to\infty} \left(\frac{f(n)}{a \cdot f(n/b)} \right) > 1.$$

Definition 8. An algorithm's **recursion tree** is an a-ary tree used to visualize its recurrence given as $T_n = a \cdot T_{n/b} + f(n)$. Calling T_n costs $f(n)$ units of "work" and results in a new problem with worst-case time $T_{n/b}$ being called "a" times. Each node in the tree represents an algorithm call, with the root node referring to the first call on a problem of size n. Given the recursion tree for some algorithm, the **time complexity** of the algorithm is the sum of the required work over all of its nodes.

The node representing a problem of time n is denoted by



where $f(n)$ is the work required, and each of the "a" lines lead to a problem of size n/b.

Lower Bounds

c.f. Devroye's printed notes for theory.

Data Structures

Definition 9. An **endogenous** data structure stores values in the structure, while an **exogenous** data structure maintains pointers to externally stored values.

1 *Divide-and-Conquer*

LECTURE 01/09

Example 1.1. Fast Exponentiation: compute x^n efficiently.

$$\begin{bmatrix} x_n \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{n-1} \\ x_{n-2} \end{bmatrix} = M^2 \begin{bmatrix} x_{n-2} \\ x_{n-3} \end{bmatrix} = \cdots = M^{n-1} \begin{bmatrix} x_1 \\ x_0 \end{bmatrix}$$

Compute M^n, where $M = \left[\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right]$:

MATRIX(n)

```
if n = 0 return [1 0; 0 1]
if n = 1 return M
else
    if n even return MATRIX(n/2)
    if n odd return MATRIX((n-1)/2) · M
```

TIME COMPLEXITY:

RAM model:	*Bit* model:
$T_n = T_{\frac{n}{2}} + 1$	$T_n = T_{\frac{n}{2}} + n^2$
$\Rightarrow T_n = \Theta(\log_2 n)$	$\Rightarrow T_n = \Theta(n^2)$

LECTURE 01/11 [3]

[3] Scribed: 2018.

Example 1.2. Chip Testing: determine state (G/B) of all chips, where tester oracle has two chips evaluate each other, i.e. 3 possible outputs: GG (both from $\mathscr{G}$ or both from $\mathscr{B}$), GB and BB (at least one from $\mathcal{B}$).[4]

[4] Given $|\mathscr{G}| > |\mathscr{B}|$ where:
$\mathscr{G}$: set of good chips;
$\mathscr{B}$: set of bad chips

ALGORITHM(n)

```
GOOD = FINDGOODCHIP(n)
do n − 1 tests against GOOD to determine state of all chips.
```

FINDGOODCHIP(n)

```
if n = 1 return the one chip.
if n even
    pair all chips and test (n/2 pairs).
    eliminate all non-GG pairs.
    eliminate the second chip of each GG pair.
    FINDGOODCHIP(remaining chips)
else if n odd
    take a random chip x and test against all other chips.
    take a majority vote:
    if ≥ (n−1)/2 G votes: x ∈ 𝒢
        return x
    else x ∈ ℬ
        eliminate x
        FINDGOODCHIP(remaining chips with n even)
```

TIME COMPLEXITY of FINDGOODCHIP(n):

$$\begin{cases} T_n \le \frac{3n}{2} + T_{\lfloor \frac{n}{2} \rfloor} \\ T_1 = 0,\ T_2 = 0 \end{cases} \Rightarrow T_n = \Theta(n)$$

Example 1.3. Karatsuba Multiplication: multiply two n-bit numbers a_n and b_n efficiently.

$$\begin{cases} a_n = (\underbrace{[010...]}_{\alpha_2}\underbrace{[...001]}_{\alpha_1}) = \alpha_1 + \alpha_2 \times 2^{n/2} \\ b_n = (\overbrace{[000...]}^{\beta_2}\overbrace{[...110]}^{\beta_1}) = \beta_1 + \beta_2 \times 2^{n/2} \end{cases}$$

$$a_n \times b_n = \alpha_1\beta_1 + \alpha_2\beta_2 + (\alpha_1\beta_2 + \alpha_2\beta_1) \times 2^{n/2}$$

where $\alpha_1\beta_2 + \alpha_2\beta_1 = (\alpha_1 - \alpha_2)(\beta_1 - \beta_2) + \alpha_1\beta_1 + \alpha_2\beta_2$.

```
MULTIPLY(a_n, b_n)
one = MULTIPLY(α_1, β_1)
two = MULTIPLY(α_2, β_2)
three = MULTIPLY(α_1 − α_2, β_1 − β_2)
return one + two + (one + two + three) × 2^{n/2} (shifting)
```

TIME COMPLEXITY:

$$T_n = 3T_{\frac{n}{2}} + 2n + 10n \quad \Rightarrow \quad T_n = \Theta(n^{\log_2 3})$$

TOOM-COOK OPTIMIZATION:

$$T_n = 5T_{n/3} + o(n) = \Theta(n^{\log_3 5})$$

Example 1.4. Convex Hull: find C.H. in order given n points in $\mathbb{R}^2$.

```
FINDUPPERHULL(1 ··· n)
FINDUPPERHULL(1, ..., n/2).
FINDUPPERHULL(n/2 + 1, ..., n).
MERGE:
    progress by x-coordinate from left to right,
    eliminate points with angle < 180°.
```

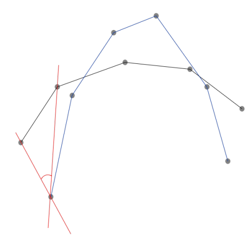


Figure 1: Diagram of the merging step. The blue and black lines are the two upper hulls.

TIME COMPLEXITY:

$$T_n = 2T_{\frac{n}{2}} + n \quad \Rightarrow \quad T_n = \Theta(n \log n)$$

LECTURE 01/16 [5]

[5] Scribed: 2018, Adam.

Example 1.5. Strassen Matrix Multiplication: matrix multiply two $n \times n$ matrices A and B.

$$A \times B = \left[\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right] \times \left[\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array}\right] = \left[\begin{array}{c|c} A_1B_1 + A_2B_2 & A_1B_2 + A_2B_4 \\ \hline A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{array}\right]$$

Use algebraic trick to reduce the 8 multiplications to 7, yielding:

TIME COMPLEXITY:

$$T_n = n^2 + 7T_{n/2} \Rightarrow T_n = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$$

LECTURE 01/18 [6]

[6] Scribed: 2018, Olivia

Example 1.6. Selection Problem: finding the kth smallest number in a set $S = \{x_1, \cdots, x_n\}$.

LECTURE 01/23

Example 1.7. Mergesort. Prove inductively that $T_n = \Theta(n \log n)$.
$T_n = n - 1 + T_{\lfloor \frac{n}{2} \rfloor} + T_{\lceil \frac{n}{2} \rceil}$, where n: number of comparisons.

Proof. Hypothesis $P(n)$: $\exists c > 0$ s.t. $T_n \leq cn \log_2 n$.

- **Base Case** $n = 0$: OK.
- **Inductive Step**: Assume $P(k)$ true up to $n - 1$.

n even:

$$\begin{aligned} T_n &\leq n - 1 + 2c\left(\frac{n}{2}\right)\log_2\left(\frac{n}{2}\right) \\ &= n - 1 + cn\log_2 n - cn \\ &\leq cn \log_2 n \qquad \text{for all } c \geq 1 \end{aligned}$$

n odd:

$$\begin{aligned} T_n &\leq n - 1 + c\left(\frac{n+1}{2}\right)\log_2\left(\frac{n+1}{2}\right) + c\left(\frac{n-1}{2}\right)\log_2\left(\frac{n-1}{2}\right) \\ &= n - 1 + c\frac{n}{2}\log_2\left(\frac{n^2-1}{4}\right) + \frac{1}{2}c\log_2\left(\frac{n+1}{n-1}\right) \\ &\leq n - 1 + c\frac{n}{2}\log_2\left(\frac{n^2}{4}\right) + \frac{1}{2}c\log_2(2) \\ &= n - 1 + c\frac{n}{2}(2\log_2 n - 2) + \frac{1}{2}c \\ &= n - 1 - c(n - \frac{1}{2}) + cn\log_2 n \\ &\leq cn\log_2 n \qquad \text{for all } c \geq 0 \end{aligned}$$

□

Example 1.8. Selection Problem Take II. Finding the k^{th} smallest element in a set of size n.

2 *Dynamic Programming*

Example 2.1. Binomial Coefficient.

Example 2.2. Travelling Salesman Problem.

LECTURE 01/25 [7]

[7] Scribed: 2018.

Example 2.3. LCS.

Example 2.4. Knapsack Problem.

Example 2.5. Optimal Binary Search Tree

- **Context**: Static trees are simpler versions of normal dynamic search trees: no adding, removing, etc. They are used for example in CPU for parsing programs (reading key words).
  The cost of the tree is defined as: $C = \sum_{i=1}^{n} w_i l_i$ where w_i are is weights of each key and l_i its level (distance from the root)
- **Goal**: Given a static set of n ordered keys with weights assigned to each, find the minimizing tree (minimize C).

ALGORITHM to find $C[1,n]$:

$C[i,j]$ = minimal optimal cost for problem with items $i \cdots j$.

$\implies$ Will need $W[i,j] = \sum_{k=1}^{j} w_k$ = weights of $i \cdots j$.[8]

[8] This can be done in $\Theta(n^2)$ time:

WEIGHTS(w_i)

for $i = 1$ to n
    $W[i,i] = w_i$
    for $j = i+1$ to n
        $W[i,j] = W[i,j-1] + w_j$

DYNAMIC PROGRAM

// Initialization
for $i = 1$ **to** n
    $C[i,i] = w_i$ // tree with only one key
    if $j < i$ // invalid tree (interval with end index > start index)
        $C[i,j] = 0$
// Recursive Step
for $size = 2$ **to** n // size of interval
    for $i = 1$ **to** n // starting index of interval
        $j = i + size - 1$
        if $j \leq n$ // to ensure a valid interval (last index max n)
            $C[i,j] = \min_{i \leq k \leq j} (C[i,k-1] + C[k+1,j] + W[i,j])$
            $root[i,j]$ = one of the r's that minimizes $C[i,j]$
return C, $root$

$$
\begin{aligned}
C[i,j] &= \sum_{\substack{i=1 \\ i \neq k}}^{j} w_i(l_i + 1) + w_{root} \\
&= \sum_{\substack{i=1 \\ i \neq k}}^{j} w_i l_i + \sum_{\substack{i=1 \\ i \neq k}}^{j} w_i + w_{root} \\
&= \sum_{i=1}^{k-1} w_i l_i + \sum_{s=k+1}^{j} w_s l_s + \sum_{t=i}^{j} w_t \\
&= C[i,k-1] + C[k+1,j] + W[i,j]
\end{aligned}
$$

$\implies$ The complexity of this algorithm is $O(n^3)$ since we have $\sum_{k=2}^{n} \sum_{m=1}^{n}$ *size* computations/runs through the loop.[9]

[9] Knuth reduced this to n^2: look up and add info.

LECTURE 01/30

Example 2.6. Matching Problem.

Example 2.7. Job Scheduling.

LECTURE 02/01 - LOWER BOUNDS [10]

[10] c.f. Devroye's printed notes.

3 *Data Structures*

LECTURE 02/06

3.1 *Lists*

LECTURE 02/08

3.2 *Trees*

types of trees
traversals & transforming btw storing representations
lung diagram

LECTURE 02/13

dictionaries

3.3 Red-Black Trees

LECTURE 02/20 [11]

[11] Scribed: 2018, Akshal

3.4 Augmenting Data Structures

LECTURE 02/22 [12]

[12] c.f. CLRS

Example 3.1. RB tree + Linked List.

Example 3.2. Order Statistic Tree.

Example 3.3. Interval Tree.

Example 3.4. Level Linked RB Tree (for fast browsing).

LECTURE 02/27 [13]

[13] Scribed: 2017 notes.

3.5 Cartesian Trees

Definition 10. A **cartesian tree** on n nodes (Figure 3.5) is a binary tree storing n tuples $(x_i, y_i)\ :\ (key,\ timestamp)$: we build a BST by inserting the nodes by increasing timestamps. The cartesian tree is therefore a *binary search tree* with respect to keys x_i, and a *heap* with respect to timestamps y_i.

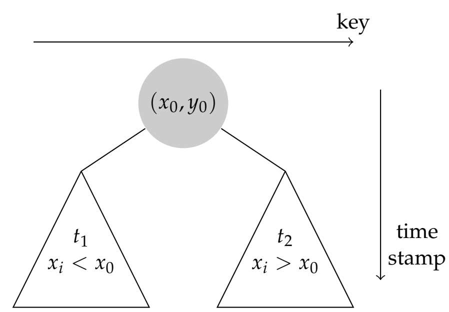


Figure 2: Diagram of a cartesian tree: (BST) keys are increasing when sweeping from left to right, and (heap) timestamps are increasing on any path from a root to a leaf.

Example 3.5. BST: inserting $x_1, \cdots, x_n$ in sequence, i.e. a cartesian tree for $(x_1, 1), (x_2, 2), \cdots, (x_n, n)$.

Example 3.6. RBST: randomly permuting all elements before adding them, i.e. a cartesian tree for $(x_1, \tau_1), (x_2, \tau_2), \cdots, (x_n, \tau_n)$ where $(\tau_1, \cdots, \tau_n)$ is a random permutation of $(1, \cdots, n)$.

Example 3.7. Treap: same concept as RBST, i.e. a cartesian tree for $(x_1, u_1), (x_2, u_2), \cdots, (x_n, u_n)$ where $u_i \in [0, 1] \in \mathbb{R}$ random.

Atomic Operations on Cartesian Trees:

i.e. FIX-ing operations: the input is (v, t), a Cartesian tree t in which only one node, v, does not satisfy the heap property.

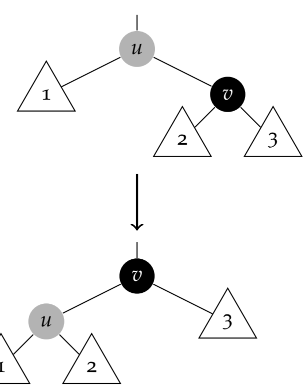


Figure 3: tree rotation done during FIX-UP and FIX-DOWN. In FIX-UP, the black node v is misplaced, it should be above the gray node u. In FIX-DOWN, the gray node is misplaced, it should be below the black one (only difference is which we are pointing to).

FIX-UP(*node*)

```
while node ≠ root && node.y < node.parent.y
    if node == node.parent.right then
        node.parent.right = node.left
        node.left = node.parent
        node.parent.left = node.right
        node.right = node.parent
```

FIX-DOWN(*node*)

```
while node is not leaf && (node.y > node.left.y || node.y > node.right.y)
    if node.right.y > node.left.y then
        node.left = node.left.right
        node.left.right = node
        node.right = node.right.left
        node.right.left = node
```

Other Operations on Cartesian Trees:

Exercise 3.8. Study how to do these (recursively) without using FIX-UP and FIX-DOWN.

1. INSERT(t,(x, y)): Inputs are a Cartesian tree t and a node (x, y) to be added to the tree. First, insert the node (x, ∞) just as you would do it in a binary search tree, implying the node will be a leaf since the heap property must be satisfied. Second, change the node to (x, y) and use FIX-UP to fix the Cartesian tree.

2. DELETE(t, *node*): Inputs are a Cartesian tree t and a pointer *node* to a node that will be removed. First, change *node.y* to ∞ and use FIX-DOWN to move the node down the tree. It will end up as a leaf so the second step is just to remove the node.

3. JOIN(t_1,t_2): Inputs are two Cartesian trees t_1 and t_2 that need to be joined into a single tree, given that all keys in t_1 are smaller than all keys in t_2. First, create a temporary node $(k, -\infty)$ such that $MAX_X(t_1) < k < MIN_X(t_2)$ and let its left child be the root of t_1 while its right child is the root of t_2. Second, delete the node and the tree will fix itself up.

4. SPLIT(t,k): Inputs are a Cartesian tree t and a value k that will split the tree into two trees that have x coordinates smaller than k and bigger than k, respectively. First, insert a temporary node $(k, -\infty)$ (it will end up as a root) and after the procedure, the left subtree and right subtree of the node are the trees we are looking for.

Expected Value Analysis for Treaps:

Define D_k to be the depth of the node with rank k, namely (k, T_k). Since the depth of a node is equivalent to the number of ancestors it has, let

$$D_k = \sum_{j \neq k} X_{jk} \text{ where } X_{jk} = \begin{cases} 1, & \text{if } j \text{ ancestor of } k \\ 0, & \text{otherwise} \end{cases}$$

So next we wish to calculate the expected value for D_k.

$$\begin{aligned}
\mathbb{E}[D_k] &= \sum_{j \neq k} \mathbb{P}((j, T_j) \text{ is an ancestor of } (i, T_i)) \\
&= \sum_{j \neq k} \mathbb{P}(T_j \text{ is the smallest of } T_j, T_{j+1}, \cdots, T_k) \\
&= \sum_{j<k} \frac{1}{k-j+1} + \sum_{j>k} \frac{1}{j-k+1} \\
&= \left(\frac{1}{2} + \cdots + \frac{1}{k}\right) + \left(\frac{1}{2} + \cdots + \frac{1}{n-k}\right) \\
&= (H_k - 1) + (H_{n-k+1} - 1) \\
&\leq 2\ln(n) \simeq 1.39 \log_2(n).
\end{aligned}$$

We will denote H_k to be the harmonic number, $\sum_{n=1}^{k} \frac{1}{n} = 1 + \frac{1}{2} + \cdots + \frac{1}{k}$. It can be approximated by $\int_i^k \frac{1}{x} dx \approx \ln(k)$ but more importantly, it can be bounded as $H_k \in [\ln(k+1), \ln(k) + 1]$.

Example 3.9. Quicksort Tree: the QUICKSORT procedure is equivalent to building a Treap (or RBST): same number of comparisons needed.

QUICKSORT(*list*, *low*, *high*)

```
i = random index between low and high
a = list[i]
if high > low then
    Partition list[low...high] so elements before index j are
    smaller than a and elements after index j are greater than a
    QUICKSORT(list, low, j − 1)
    QUICKSORT(list, j + 1, high)
```

EXPECTED COMPLEXITIES FOR QUICKSORT:

1. Let C_n = # of comparisons $= \sum_{i=1}^{n} D_i$.

$$\begin{aligned}
\mathbb{E}[C_n] &= \sum_{i=1}^{n} E[D_i] \;=\; \sum_{i=1}^{n} (H_i + H_{n-i+1} - 2) \;=\; 2\sum_{i=1}^{n}(H_i - 1) \\
&= 2\sum_{i=1}^{n} H_i - 2n \;=\; 2\sum_{i=1}^{n}\sum_{j=1}^{i}\frac{1}{j} - 2n \;=\; 2\sum_{j=1}^{n}\frac{1}{j}\sum_{i=j}^{n} 1 - 2n \\
&= 2\sum_{j=1}^{n}\frac{1}{j}(n-j+1) - 2n \;=\; 2(n+1)H_n - 4n \\
&\simeq 2n\ln(n) \;=\; 1.386294\ldots n\log_2(n)
\end{aligned}$$

2. Operations INSERT, DELETE, JOIN and SPLIT are all $O(\log n)$.

LECTURE 03/01

3.6 Priority Queues

Definition 11. A **priority queue** is an abstract data structure storing items with keys, supporting operations INSERT, DELETEMIN, DELETE, DECREASEKEY and sometimes SEARCH. Implementation possibilities are red-black trees, standard heaps or **tournament trees**. It can be applied to (1) sorting, (2) operating systems, and (3) Discrete Event Simulation (DES).

Definition 12. A **tournament tree** is a complete binary tree where all items are leaves (n) and all internal nodes represent matches between two items, where they store pointers to winners of the match ($n-1$). It is implemented as an array of size $2n-1$.[14]

[14] insert figures

All the operations MAKETREE$()$, UPDATE(x, i), INSERT(x) and DELETEMIN$()$ can be performed in $n-1$ comparisons. c.f. detailed notes for algorithms.

4 Strings

4.1 Information Theory

LECTURE 03/13

Definition 13. The **compression ratio** $R = \frac{\text{\# bits in outbut } B}{\text{\# bits in input } A}$.

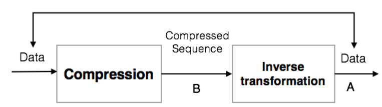


Figure 4: Communication system diagram: compression and inverse transformation. An input $x_1, \ldots, x_n$ is compressed "coded" into bits $b_1, \ldots, b_N$. For ease of communication and storage, we want $N \ll n$, but the possible smallest N is limited by the amount of choices in the input.

SHANNON's postulate: every possible input sequence that may have to be compressed has a given probability p_i s.t. $\sum_i p_i = 1$. If the i-th input sequence is compressed into an output sequence of length l_i, then the expected length of the output bit sequence is $\sum_i p_i l_i$.

1. Find a binary tree that minimizes $\sum_i p_i l_i$: the **Huffman tree**.

2. Show that $\min \sum_i p_i l_i \geq \epsilon$ where $\epsilon \equiv \sum_i p_i \log_2 \frac{1}{p_i} = -\sum_i p_i \log_2 p_i$.

Definition 14. The **Huffman Tree** has two conditions:

1. the smallest probability p_i must be the farthest away from root

2. there are no one child nodes.

MAKEHUFFMANTREE$((i, p[i])$ pairs$)$

MAKENULL(PQ) // create empty PQ (min-heap)
for $i = n$ to $2n-1$ // add all elements to PQ
    $left[i] = right[i] =$ NIL
    INSERT$((i, p[i]),$ PQ$)$
for $i = n-1$ down to 1 // two min elements
    $(a, p) =$ DELETEMIN(PQ) $(b, q) =$ DELETEMIN(PQ)
    $left[i] = a$ $right[i] = b$
    INSERT$((i, p+q),$ PQ$)$

This greedy algorithm takes $O(n \log n)$.

Theorem 15. $\min \sum_i p_i l_i \geq \epsilon$, *where* $\epsilon \equiv \sum_i p_i \log_2 \frac{1}{p_i}$.

Proof. Recall Kraft's Inequality $\sum \frac{1}{2^{l_i}} \leq 1$ and the fact $\log_e x \leq x - 1$ (by Taylor).

$$\begin{aligned}
\sum_i p_i l_i &= \sum_i pi \log_2 2^{l_i} = \sum_i -p_i \log_2 p_i + p_i \log_2 p_i + p_i \log_2 2^{l_i} \\
&= \sum_i p_i \log_2 \frac{1}{p_i} + \sum_i p_i \log_2(p_i 2^{l_i}) = \epsilon + \sum_i p_i \log_2(p_i 2^{l_i}) \\
&= \epsilon + \sum_i p_i \frac{\log_e(p_i 2^{l_i})}{\log_e 2} = \epsilon - (\log_2 e) \sum_i p_i \log_e \frac{1}{p_i 2^{l_i}} \\
&\geq \epsilon - \log_2 e \sum_i p_i \left(\frac{1}{p_i 2^{l_i}} - 1 \right) = \epsilon - \log_2 e \left(\sum_i \frac{1}{2^{l_i}} - \sum_i p_i \right) \\
&= \epsilon - \log_2 e((\leq 1) - 1) \geq \epsilon
\end{aligned}$$

□

Definition 16. The **Shannon-Fano** code has $\sum_i p_i l_i \leq \epsilon + 1$. It is constructed by setting $l_i = \lceil \log_2 \frac{1}{p_i} \rceil$ for each p_i.

Proof. Recall $\sum_i \frac{1}{2^{l_i}} \leq 1$.

$$\sum_i p_i l_i = \sum_i p_i \lceil \log_2 \frac{1}{p_i} \rceil \leq \sum_i p_i \left(\log_2 \frac{1}{p_i} + 1 \right) \leq \epsilon + 1$$

□

Lecture 03/15[15]

[15] Scribed, 2018.

Definition 17. The **Lempel-Ziv (LZ) Method** is a compression method that generalizes the Hoffman tree to any continuously updating alphabet.

Definition 18. A **Digital Search Tree (DST)** is a k-ary tree (k-alphabet) where each node is a piece of the input string (a previously identified piece plus one new symbol), and each edge is a symbol.[16]

[16] c.f. full scribed notes for diagrams.

The CODER builds a DST in $O(n)$ time where n is the number of input symbols. Observing that the LZ sequence stores the DST via parent pointers, the DECODER uses an array representation to reconstruct the original string in $O(n)$ time as well.

4.2 *Data Structures*

LECTURE 03/16

Items 1. - 3. are used for strings $x_1, \ldots, x_n$ where $x_i = (x_{i1}\ x_{i2}\ \ldots)$ s.t. $x_{ij} \in A$ (alphabet); and items 4. - 6. are used for text $T = (t_1\ \ldots\ t_n)$ where $t_i \in A$.

1. TRIE: a tree-based data structure for storing strings, where each string is a path in the k-ary tree (each edge is a letter from the k-alphabet). The leaf stores the string's index.

Data structures used for a trie:

(a) array of children (time efficient)

(b) linked list of children (space efficient)

2. PATRICIA TRIE: a compact trie, where all the nodes with one child are collapsed.

Show that for a k-ary PATRICIA, the number of nodes is less than $2n - 1$

3. DIGITAL SEARCH TREE (DST): a k-ary tree where one inserts $x_1, x_2, \ldots, x_n$ in turn, and each string occupies the first available slot in a method like a binary search tree.

4. SUFFIX TRIE: a trie containing the n suffixes of the given text T.

$$
\begin{aligned}
T = T[1] \ \ldots \ T[n] &= x_1 \\
T[2] \ldots T[n] &= x_2 \\
&\ldots \\
T[n] &= x_n
\end{aligned}
$$

5. SUFFIX TREE: a PATRICIA version of the suffix trie.

6. SUFFIX ARRAY: a sorted array containing a lexicographical ordering of the suffixes, i.e. with leaf numbers in the suffix trie seen in preorder. One can then locate strings by binary search $O(\log n)$.

4.3 String Matching

Algorithm by Knuth, Morris and Pratt given a text T of length n and pattern P of length m where $m \ll n$ which runs in time $O(m + n) = O(n)$ as opposed to the naive $O(mn)$.

KMP ALGORITHM(T, P)

preprocess P in time $O(m)$
actual string matching with T in time $O(n)$