Algorithmics I
Dr.Gethin Norman

Joao Almeida-Domingues[*]

University of Glasgow

October 1st, 2020 – December 2nd, 2020

Contents

These lecture notes were collated by me from a mixture of sources , the two main sources being the lecture notes provided by the lecturer and the content presented in-lecture. All other referenced material (if used) can be found in the *Bibliography* and *References* sections.

The primary goal of these notes is to function as a succinct but comprehensive revision aid, hence if you came by them via a search engine , please note that they're not intended to be a reflection of the quality of the materials referenced or the content lectured.

Lastly, with regards to formatting, the pdf doc was typeset in LaTeX, using a modified version of Stefano Maggiolo's class

[*]2334590D@student.gla.ac.uk

1 Introduction

Lecture 1
October 1st, 20

1.1 *Topics*

- Sorting and Tries
- Graphs and graph algorithms
- Strings and text algorithms
- An introduction to NP completeness
- A (very) brief introduction to computability

It is assumed that you're familiar with the following from year 2

- Java
- fundamentals of graph theory
- sets, relations, functions
- proofs
- ADTs, lists, arrays, queues, binary trees, red-black trees, b-trees, hash tables
- algorithms analysis
- insertion-sort, selection-sort, bubblesort, merge-sort, quicksort, heapsort, counting-sort, radix sort
- linear search, binary search

1.2 *Revision*

big O notation , polynomial algos , exponential algos , stack , queues , asymptotic behaviour

1.1 remark. See ADS2 & AF2 notes for a more in-depth explanation of these topics

Algorithm Analysis

Time Complexity as a function of input size, i.e given a certain input how long will it take for the algorithm to terminate. Because we are interested in what will happen as the input grows, so that we can have a *guarantee* on the algorithm's performance, we are usually concerned with the *worst-case* scenario

1.2 remark. This asymptotic behaviour is generally expressed using the *Big O* $\mathcal{O}$ notation

1.3 remark. For some algorithms is is also important to consider *space complexity*

Big O

$$f(n) = \mathcal{O}(g(n))$$

We say that f grows *strictly* slower than g. Formally, for two functions $f, g \in \mathbb{N}$ we have

$$|f(n)| \leq |cg(n)| \forall n \geq N, c \in \mathbb{R}, N \in \mathbb{Z}$$

Usually f is a complex function which is not known precisely and g is a known function (linear, quadratic etc.) which essentially *bounds f*

We want to be as precise as possible, so we use the *"tightest"* bound as possible, e.g whilst n^3 is also an upper bound on a linear time n algorithm, if we know that the algorithm grows no faster than n^2 then that is the lowest-/tightest bound, and the one we should use

ADT - Stack

Stacks order of removal is *LIFO* and can be represented as an an array or a linked list, all operations are $\mathcal{O}(1)$

Basic Ops:

- `create`
- `isEmpty`
- `push`
- `pop`

ADT - Queue

Stacks order of removal is *FIFO* and can be represented as an an array or a linked list, all operations are $\mathcal{O}(1)$

1.4 remark. the array must wraparound, i.e once `end == a.len() ; end = 0`

Basic Ops:

- `create`
- `isEmpty`
- `inser`
- `delete`

Priority Queue

– as an unordered list (insert is $0(1)$ while delete is $0(n)$)
– as an ordered list (insert is $0(n)$ while delete is $0(1)$)
– as a heap (insert and delete are $0(\log n)$)

2 Sorting and Tries

🔍 comparison based algorithms , Decision tree , Information-Theoretic , Lower Bound Ω , Radix Sort

2.1 remark. Even though Quicksort has a worst case running time of $O(n^2)$, in practice it is the fastest sorting algorithm of the ones we've seen with a running time of $O(n \log n)$

2.2 definition. **Comparison Based Sorting** algortihms can only access the input elements by the comparisons $A[i] < A[j], A[i] \leq A[j], A[i] = A[j], A[i] \geq A[j], A[i] > A[j]$

2.3 example. Insertion-Sort , Quicksort, Merge-Sort, Heapsort
mayr˙2020

2.4 definition. **Decision Tree Model** a tree where at each node two elements are compared so that each node represents the output of a comparison-based algorithm S on an input array $A = \{A[1], \ldots, A[n]\}$, and a full execution is represented by a path from the root to a leaf node

For S, we have :

$$\begin{aligned} C_S(n) = &\text{ worst-case number of comparisons performed} \\ &\text{by } S \text{ on an input array of size } n. \end{aligned}$$

2.5 theorem. *For all comparison based sorting algorithms S*

$$C_s(n) = \Omega(n \log n)$$

where Ω represents the lower bound, i.e best possible running time

2.6 remark. We assume that the rare case where $A[i] = A[j]$ does not happen, for simplicity. We can do this because we are looking for a lower bound not an upper bound

Proof.

2.7 lemma. *For every $n, C_s(n)$ is the height of the decision tree of S on inputs n*
This follows from the fact that the longest path from the root to a leaf is the maximum number of comparison that S will do on an input of length n

2.8 lemma. *Each permutation of the inputs must occur at least one leaf of the decision tree*
This is necessarily true if the algorithm works properly for all inputs, since the number of leaf nodes in the tree must be at least the number of possible outcomes of the algorithm
The simplified decision tree is a binary tree, which means that for a binary tree of height h we have at most 2^h leaves. Hence,

$$\begin{aligned} n! &\leq \textit{number of leaves of decision tree} \\ &\leq 2^{\textit{height of decision tree}} \\ &\leq 2^{C_S(n)} \end{aligned}$$

Hence, we have that the height of the tree, and therefore one possible execution of the algorithm, is at least $\log(n!)$ *we have found our upper bound. We now observe that* $n^{n/2} \leq n! \leq n^n$

Which tells us that,

$$\lg n! \geq \log\left(n^{n/2}\right) = (n/2)\log n = \Omega(n\log(n))$$

2.1 *Radix Sorting*

Radix sort uses a different approach than comparison based algos and can therefore achieve $O(n)$ complexity by exploiting the structure of the items being sorted

this does imply that it is less versatile

Assume that the items to be sorted can be treated as bit-sequences of length m and let $b|m$. We can then label each bit position from $\{0, 1, \ldots, m-1\}$.

The algorithm uses $\frac{m}{b}$ iterations where in each iteration the times are distributed into 2^b *"buckets"*/lists, which are themselves labelled from $\{0, \ldots, 2^b - 1\}$ * .

* *equivalently,* $\overbrace{00\ldots0}^{\textit{length } b}, \overbrace{11\ldots1}^{\textit{length } b}$

During the i^{th} iteration an item is placed in the list corresponding to the integer represented by the bits in position $b \times i - 1, \ldots, b \times (i-1)$

2.9 example. For $b = 4, i = 2$, consider the bits in position $7, \ldots, 4$

$$\text{item } = 00101001[0011]0001$$

0011 represents the integer 3, so the item is placed in the bucket labelled 3 or 0011

Algorithm Step-by-Step

1. Find the max value so that we know how many bits we need for our binary enconding
2. Write each element in binary
3. Calculate m, choose b and find the required number of 2^b buckets and i iterations
4. Sort the items into appropriate buckets by matching the integer represented by the bits at position $b \times i - 1, \ldots, b \times (i-1)$ with the corresponding bucket's integer label
5. Concatenate the buckets and repeat $i - 1$ times

2.10 example. For the sequence $15, 43, 5, 27, 60, 18, 26, 2$

See here

Pseudocode Implementation

```
// assume we have the following method which returns
// the value represented by the b bits of x when starting at position pos
private int bits(Item x, int b, int pos)

// suppose that:
// a is the sequence to be sorted
// m is the number of bits in each item of the sequence a
// b is the block length of radix sort

// number of iterations required for sorting
int numIterations = m / b;

// number of buckets
int numBuckets = (int) Math.pow(2, b);

// represent sequence a to be sorted as an ArrayList of Items
ArrayList < Item > a = new ArrayList < Item > ();

// represent the buckets as an array of ArrayLists
ArrayList < Item > [] buckets = new ArrayList[numBuckets];

for (int i = 0; i < numBuckets; i++) buckets[i] = new ArrayList < Item >
    ();

for (int i = 1; i <= numIterations; i++) {
  // clear the buckets
  for (int j = 0; j < numBuckets; j++) buckets[j].clear();
  // distribute the items (in order from the sequence a)
  for (Item x: a) {
    // find the value of the b bits starting from position (i-1)*b in x
    int k = bits(x, b, (i - 1) * b); // find the correct bucket for item x
    buckets[k].add(x); // add item to this bucket
  }
  a.clear(); // clear the sequence

  // concatenate the buckets (in sequence) to form the new sequence
  for (j = 0; j < numBuckets; j++) a.addAll(buckets[j]);
}
```

Correctness

For the algorithm to be correct we need to know that $\forall\ x, y | x < y$ in the last iteration j , x must precede y. Since $x < y$ and j is the last iteration that x and y bits differ, then the relevant bits of x must be smaller than those of y. Hence, x goes into an *"earlier"* bucker than y and so follows that x precedes y in the sequence after this iteration

Finally, since j is the last iteration where bits differ it follows that in all later iterations x, y go in the same bucket so their relative order is unchanged

Complexity

We have that $i = \frac{m}{b}$ and $B = 2^b$, and during each of the iterations we go through the list and allocate items to corresponding buckets so this takes $O(n)$ time because you just need to take each item and look at the corresponding bits and move it to the bucket. However, there are 2^b buckets, hence concatenating them will take $O(2^b)$ time. Hence, overall

$$O(\frac{m}{b}(n + 2^b)) = O(n), \text{ since } m, b \text{ are constants}$$

2.11 remark. there is a time-space trade-off involving our choice of b. larger b implies less iterations, but more buckets

2.2 *Tries Data Structure*

A trie is a tree-based data structure for storing strings in order to support fast pattern matching

2.12 remark. *Tries* are to trees what Radixsort is to comparison-based algos

2.13 remark. The main application for tries is in information retrieval. Indeed, the name comes from the word "retrieval"

2.14 definition. Standard Trie goodrich˙tamassia˙2011 Let S be a set of s strings from alphabet L, such that no string in S is a prefix of another string. A standard trie for S is an ordered tree $T$with the following properties:

1. Each node of T, except the root, is labelled with a character of L
2. The ordering of the children of an internal node of T is determined by a canonical ordering of the alphabet L.
3. T has s external nodes, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to an external node y of T yields the string of S associated with y.

Implementation

Tries can be implemented using an array of pointers to represent the children of each node, or via a linked list to represent the children of each node.

2.15 remark. There is a space/time trade-off between the two

The list implementation essentially consists of a node with at most 4 elements:

- label
- word flag , a T/F value indicating whether that node is part of a word

0-2 pointers, pointing to a child and/or sibling

Java implementation Trie Java implementation Node

Search Pseudocode

2.16 remark. the complexity of trie operations is essentially linear in the string length and is almost independent of the number of items

```
// searching for a word w in a trie t
Node n = root of t; // current node ( start at root )
int i = 0; // current position in word w ( start at beginning)

while (true) {
        if (n has a child c labelled w.charAt(i)) {
// can match the character of word in the current position
                if (i == w.length()−1) { // end of word
                        if (c is an 'intermediate' node) return "absent";
                        else return "present";
                } else { // not at end of word
                        n = c; // move to child node
                        i++; // move to next character of word
                }
        } else return "absent"; // cannot match current character
}

// inserting a word w in a trie t
Node n = root of t; // current node ( start at root )
for (int i=0; i < w.length(); i++){ // go through chars of word
        if (n has no child c labelled w.charAt(i)){
// need to add new node
                create such a child c;
                mark c as intermediate;
        }
n = c; // move to child node
}
// mark n as representing a word;
```