Algorithms and Programming Techniques
COMP3121 UNSW

Hussain Nawaz
hussain.nwz000@gmail.com

2022T2

Contents

1 Introduction - Revision

1.1 Rates of Growth

The Problem To analyse algorithms, we need a way to compare two functions representing the runtime of each algorithm. However, comparing values directly presents a few issues.

- Outliers may affect the results.
- One algorithm may be slower for a set period of time, but catch up after a while.
- The runtime of an algorithm may vary depending on the implementation or the architecture of the machine where, some instructions are faster than others.

Asymptotic Growth For algorithms, we often prefer to refer to them in terms of their asymptotic growth in runtime, with relation to the input size.

A function that quadruples with every extra input will always have a greater runtime than one that increases linearly with each new input, for some large enough input size.

Big O Notation We say $f(n) = O(g(n))$ if there exists a positive constants C, N such that

$$0 \leq f(n) \leq Cg(n) \quad \forall n \geq N.$$

We may refer to $g(n)$ to be the asymptotic upper bound for $f(n)$.

Big Omega Notation We say $f(n) = \Omega(g(n))$ if there exists positive constants c, N such that

$$0 \leq cg(n) \leq f(n) \quad \forall n \geq N.$$

Then, $g(n)$ is said to be an asymptotic lower bound for $f(n)$. It is useful to say that a problem is at least $\Omega(g(n))$.

Landau Notation $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$.

There are strict version of Big O and Big *Omega* notations; these are little o and little ω respectively.

We say $f(n) = \Theta(g(n))$ if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

That is, both f and g have the same asymptotic growth.

Logarithms Logarithms are defined so that for $a, b > 0$ where $n \neq 1$, let

$$n = \log_a b \Leftrightarrow a^n = b.$$

They have the following properties:

- $a^{\log_a n} = n$

- $\log_a(mn) = \log_a(m) + \log_a(n)$
- $\log_a(n^k) = k \log_a(n)$

By the change of base,

$$\log_a(x) = \frac{\log_b(x)}{\log_b(a)}.$$

As such, the denominator is constant in terms of x and so all log bases are equivalent under Big Theta notation.

1.2 Assumed Data Structures

Arrays We assume static arrays (though, it is possible to extend to dynamic arrays).

- We assume random-access in $O(1)$
- Insert / delete $O(n)$
- Search: $O(n)$ - $\log n$ if sorted

Linked Lists We assume the linked lists are doubly-linked since the $2\times$ overhead is negligible.

- Accessing next / previous: $O(1)$
- Insert / delete to head or tail: $O(1)$
- Search: $O(1)$.

Stacks Last in, first out.

- Accessing top: $O(1)$
- Insert / delete from top: $O(1)$

Queue First in, first out.

- Access front: $O(1)$
- Insert front: $O(1)$
- Delete front: $O(1)$

Hash Tables Store values by their hashed keys. Ideally, no two keys will hash to the same value, however this may not be guaranteed.

- Search is expected to be $O(1)$ however, in the worst case, we expect to have to search through all the values in $O(n)$.
- Insertion is expected to be $O(1)$ however, in the worst case, we expect to have to search through all the values in $O(n)$.
- Deletion follows the same pattern of $O(1)$ expectation and $O(n)$ worst case.

Binary Search Trees We store (comparable) keys (or key-value pairs) in a binary tree, where each node has at most two children, the left and right. The value of a node must be greater than the values of all its children in its left sub-tree and greater than those in the right sub-tree.

- In the best case, the height of the tree is $h = \log_2 n$. Such a tree is *balanced.* in the worst case however, the tree is a long chain of height n.
- The average search is $\log n$. If the tree is self-balancing then search and other operations are guaranteed to be i $\log n$.

Binary Heap A max-heap is such that the value of a node is greater than or, equal to the value of all its children. A min-heap follows the same principle but in reverse.

- Finding the maximum involves finding the top item in $O(1)$
- Deleting the top item will also require re-balancing.
- Re-balancing the heap requires $\log n$ time.
- Insertion also requires $\log_n$ time.

1.3 Algorithms

Linear Search Given an array A of n integers. We may determine if a value is inside A, by searching through the array linearly. This occurs in $O(n)$.

Sorted Arrays - Binary Search Given a sorted array A of n integers that are sorted by value. We may determine if a value is inside A, by binary search. We pick the midpoint of the A. If the midpoint m is the value desired, we return. Otherwise, we continue to recursively search through the array by halving the search space. If the m is greater than the value we desire, then we search the right sub-array of the array; otherwise, we search the left sub-array of the array

Decision Problems and Optimisation Problems Decision problems are of the form

Given some parameters X, can you ...

Optimisation problems are of the form

What is the smallest X for which you can ...

Comparison Sorting We are given an array of n items. We may sort the array in $O(n \log n)$ at best.

Asymptotic Lower Bound on Comparison Sorting There are $n!$ permutations of the array, of which there is 1 correct sort. If we do k comparisons, then we are able to check 2^k permutations of results from these comparisons and so, can at most, distinguish 2^k permutation.

Thus, we need to perform comparisons such that $2^k \geq n!$. That is, $k \geq \log_2(n!)$. Therefore,

$$k \geq \frac{n}{2} \log_2 \left(\frac{n}{2} \right)$$

1.4 Non-Comparison Sort

Counting Sort Create another array B of size k where k is the maximum size of an element. We iterate through the array and count the number of times each element occurs, iterating the index of B that is equivalent to the value of our current element.

The time complexity is $O(n + k)$ where $O(k)$ space is required.

Bucket Sort Distribute the items into buckets $1, \ldots, k$ such that each bucket contains a range of items such that all items in bucket 1 are less than all the items in bucket 2 and so on. We may sort within that bucket with any algorithm and then concatenate all the buckets together.

Radix Sort We assume an array A of n keys, of k symbols.

We bucket the keys by their first symbol. Then, within each bucket we also classify those items by the next symbol recursively. This happens in $O(nk)$.

LSD Radix Sort Sort all keys by their last symbol, then sort all keys by their second last symbol, and so on. The sorting algorithm at each stage must be stable.

The time is $O(nk)$ with space complexity of $O(n + k)$.

1.5 Graphs

2 Algorithm Analysis

2.1 Stable Matching Problem

Hospital's Stable Matching Problem Suppose there exists n doctors and hospital. Each hospital wants to hire exactly one new doctor and the hospital ranks the priority of the doctors they would prefer and each doctor also lists their preferences for the hospitals they'd like to go to.

For a stable match, we would like to create an allocation where no group would be happy to trades their allocations with each other.

Naive Solution The naive method computes all possible allocations and then picks one that is stable.

This runs in $O(n!)$ time which is not preferable.

Fun Fact about Factorials - Sterling's Approximation

$$n! \approx \left(\frac{n}{e}\right)^n$$

Gale - Shapely Algorithm - Assumptions

- Produces pairs in stages, with possible revisions.
- A hospital which has not been paired, will be called *free*.
- Hospitals will offer jobs to doctors, who will decide whether to accept a job or not. Doctors may renege.
- All hospitals start off as free.

Gale - Shapely - Solution While there exists a free hospital which has not offered jobs to all doctors, pick a free h and have it offer a job to the highest doctor d on its list to whom they have not offered a job yet.

If no one has offered d a job yet, they will always aspect the pair h, d.

Otherwise, if they're already on the pair h', d then if h is a higher preference than h' the doctor will renege h' for h to form (h, d). Otherwise, the hospital will seek another person to offer a job.

Proving Termination in n^2 In all rounds of the Gale-Shapely algorithm, the hospital will offer a job to one doctor. They may offer a job to each doctor only once. Thus, the hospital may only make n offers. There are n hospitals each making n offers so, the algorithm must terminate in n^2 rounds or less.

Proving Correctness of Gale-Shapely If the while loop has terminated with h still free, that implies that h has offered a job to everyone. Since the only way to get declined is if another, more appealing hospital has offered a job to other doctors.

If there are no more doctors available, then n other hospitals must have offered a job to n doctors. However this would imply there are $n+1$ hospitals in total, which is a contradiction.

Hospitals will be happy as they start with their highest preference first and, doctors will move up to accept their most preferred offer. Thus, there cannot exists a circumstance where both a hospital and doctor would like to swap.

3 Divide-and-Conquer Method

3.1 Counting Inversions

Brute Force Brute force will take $O(n^2)$ time. Any method of looking at entries one by one will have have at worst, a quadratic time complexity since there can be $\frac{n^2}{2}$ inversions.

Divide and Conquer Method We can split the array into two equal parts A_{hi} and A_{lo}.

Then, for all of the elements in A_{hi} and the total inversions is the number of elements in A_{lo} that are less than the elements of A_{hi}.

We may sort both A_{hi} and A_{lo}. Then, we seek to merge the arrays together. Every time we pull an element from A, we add the number of elements remaining in A_{lo} to the total number of inversions as, all of those elements are greater than our current element from A_{hi} but to the left of our current element.

Time Complexity of Divide and Conquer Method The divide and conquer method will the same time complexity as merge sort and thus runs in $\Theta(n)$.

3.2 The Master Theorem

Setup Master Theorem Let $a \geq 1$ be and integer and $b > 1$ be a real number, $f(n) > 0$ be a non-decreasing function defined on the positive integers. Then, $T(n)$ is the solution of the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

Then, we define the critical exponent $C^* = \log_b(a)$ and the critical polynomial n^{c^*}.

Master Theorem

1. If $f(n) = O(n^{c^*})$ for some $\epsilon > 0$ then, $T(n) = \Theta(n^{c^*})$.
2. If $f(n) = \Theta(n^{c^*})$ then, $T(n) = \Theta(n^{c^*}) \log n$.
3. If $f(n) = \Omega(n^{c^*+\epsilon})$ for some $\epsilon > 0$ and, for some $c < 1$, and some n_0,

   $$af(\frac{n}{b}) \leq cf(n)$$

   holds for all $n > n_0$ then, $T(n) = \Theta(f(n))$.

If the conditions above do not hold then, the master theorem is not applicable.

3.3 Integer Multiplication and Addition

Notation and Basic Methodology We let n be the number of bits in the integer. Addition occurs by moving from the least to most significant bit, adding each bit at a time in $\Theta(n)$. Multiplication is much of the same but, in $O(n^2)$.

The Karatsuba Trick This happens in $O(n^{\log_2 3})$.

4 The Greedy Method

4.1 When greed pays off - foundations of the Greedy Method

What is a Greedy Problem A greedy algorithm will divide a problem into stages and rather than exhaustively searching through all combinations of options in all stages, it only considers the choice that is the best for the current stage.

The idea is that the search space is reduced however, it is not necessarily given that the solution is found using a greedy point.

Proofs of Greedy Algorithm There are two main methods of proof.

1. **Greedy Stays Ahead:** This proves that at every stage, no other algorithm can do better than the proposed algorithm.
2. **Exchange Argument:** Consider an optimal solution and gradually transform it to the solution found by the proposed algorithm without making it any worse.

These methods are analogous to proof by induction and contradiction respectively.

4.2 Activity Selection problem

Problem Statement There is a list of n activities with starting times s_i. and finishing time f_i. Schedule the activities such that no two activities overlap. Maximise for the total number of activities.

Solution Among the activities that do not conflict with the previously chosen activities, choose the activity with the earliest end-time. Ties may be broken arbitrarily.

Proof Correctness is proved with the *exchange argument* to show that any optimal solution can be transformed into our greedy solution.

1. Find the first place at which the optimal solution violates the greedy choice.

2. Replace the activity chose with the greedy choice. Clearly, the number of activities is the same. Also, we know there are no conflicts that have been created because, we are working left-to-right and have chosen the first conflict. So, no conflict happens before the start of the greedy's choice. Also, the finish time of greedy is no greater than the finish time of the optimal solution. As such, there is no conflict on the right-side of the greedy choice either.

Complexity We can sort using the finishing time as the key in $n \log n$. Then, loop through all the activities linearly for a total time of $O(n \log n)$.

4.3 Job Lateness

Problem At a start time T_0 and a list of n jobs with duration times t_i and deadlines t_i.
Assume that only one job can be done at a time and all jobs need to be completed

Solution Ignore the duration. Then, choose jobs in terms of ascending deadlines.

Proof Consider an optimal solution. We say i, j is an inversion if i is scheduled before j but, j is scheduled after i.

4.4 Huffman Codes

Array Merging Problem Are given n sorted arrays of different sizes. May only merge 2 at a time.

Huffman Code Given a set of symbols, encode these symbols into a binary string that can be decoded unambiguously.

Naive Solution: Ascii Designate each symbol as a character and assign a unique integer to that character. Given n characters, you require $\lceil n \log n \rceil$ bits. Wastes lots of space.