Algorithms and Data Structures

Roudranil Das

PDSP Aug-Nov 2022
Roll No: MDS202227
roudranil@cmi.ac.in

November 7, 2022

Contents

1 October 6, 2022

1.1 Orders of magnitude

- When commparing $t(n)$ focus on orders of magnitude
- Ignore constant factors
- $f(n) = n^3$ eventually grows faster than $g(n) = 5000n^2$

Can we have some **upper bounds**?

- $f(x)$ is said to be $O(g(x))$ if we can find constants c and x_0 such that $c \cdot g(x)$ is an upper bound for $f(x)$ beyond x_0
- $f(x) \leq c \cdot g(x) \forall x \geq x_0$
- Obviously $1 \leq \log(n) \leq \sqrt{n} \leq n \leq n\log(n) \leq n^2 \leq 2^n \leq n!$

Can we have some **lower bounds**?

- $f(x)$ is said to be $\Omega(g(x))$ if we can find constants c and x_0 such that $c \cdot g(x)$ is a lower bound for $f(x) \forall x \geq x_0$
- Typically we try to establish lower bounds for a problem rather than an individual algorithm.
  - If we sort a list by comparing elements and swapping them we require $\Omega(n\log(n))$ comparisons.
  - This is independent of the algorithm we use for sorting.

Can we have some **tight bounds**?

- $f(x)$ is said to be $\Theta(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x))$.

1.2 Some useful properties

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n)+f_2(n)$ is $O(\max(g_1(n), g_2(n)))$. This means that the most expensive stage determines the cost of the algorithm.

What are we measuring actually? Useful to describe asymptotic worst case running time. What matters to us is the size of the input, for example, the size of the data structures, but not the size of the actual elements of the data structure.

However this does become important later on. For example in the multiplication of large numbers. If we are considering $a \times b$ then the cost grows with a and b. Here the cost depends on the number of digits. If the number is n, then the number of digits is $\log(n)$. This becomes the cost.

2 October 11, 2022

Analysis should be independent of the underlying hardware. What are our typical basic operations: comparing values or assigning a value to a variable. What about exchanging a pair of values? All these take only a constant number of operations and so we can ignore them.
What about the input size?

- size of the list or array
- number of objects we want to rearrange
- number of vertices and number edges in a graph
- in a number theory problem, the number of digits

What inputs should we consider?

- the performance varies across the input cases
- ideally we want the average behaviour: difficult to compute, average over what? are all inputs equally likely? We need a prob distribution over all inputs
- instead consider the worst case input - input forces algo to take the longest possible time - minimise that time and we may be good

2.1 Searching in a list

Our goal is to find whether an element is present in a list `l`. Our naive approach is to go through all the elements one by one and find if it is there (linear search). This is no doubtly a very slow algo ($O(n)$). How do we know that? In the worst case the element is not present so in that case we need to go through all the n elements.

A good alternative would be binary sort: we take a sorted list and then find the midpoint and check which side of the midpoint the element we are looking for lies, take that half of the list and then repeat the procedure. Here the underlying data structure is actually important. If the underlying data structure is an array or a python list which behaves like an array then the time complexity is nothing but $\log(n)$. But if we take

the underlying data structure to be a linked list - then that basically makes no sense to sort the array because in order to get to an element in a linked list we need to traverse all the elements before that. So essentially we would be traversing all the elements so it would be $O(n)$.

Calculation of time complexity:

- if $n = 0$ we exit so $T(0) = 1$
- if $n > 0$ then $T(n) = T(n//2) + 1$
- We solve this by expanding the rhs to ultimately get $T(n) = T(n//2^k) + k$
- If for convenience sake we assume that $n = 2^k$ then we have $T(n) = T(1) + k = 2 + k = 2 + \log_2(n)$
- Ignoring constants and generalising the above assumption we get $T(n) = \log_2(n) = \log(n)$ as changing base of logarithm will atmost change by a multiplicative constant.

2.2 Sorting problems

Definitely sorting a list makes many other problems easier, like binary search, finding the median, checking for duplicates and so on. How do we sort a list?

2.2.1 Naive sorting (Selection sort)

- Find the minimum element from the entire list and move it to a different list
- Repeat with the remaining elements in the list
- Avoid using a new list:
  - swap the minimum element with the first position
  - swap the second minimum with the second position
  - continue

Obviously the time complexity is $T(n) = O(n^2)$

2.2.2 Insertion sort

- start building a new sorted list
- pick next element and insert it into the sorted list
- an iterative formulation
  - assume that `l[:i]` is sorted
  - insert `l[i]` in `l[:i]`
- A recursive formulation
  - inductiively sort `l[:i]`
  - insert `l[i]` in `l[:i]`

3 October 18, 2022

3.1 Sorting Algorithms contd

3.1.1 Analysis of insertion sort

Correctness of the algorithm follows from the invariant. Efficiency of the iterative form:

- outer loop n times
- inner loop i steps to insert in the worst case
- $T(n) = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$. Hence $T(n) = O(n^2)$

Efficiency of the recursive form:

- For input of size n let $TI(n)$ be time to insert and $TS(n)$ be the time to sort.
- then $TI(0) = 1$, $TI(n) = TI(n-1) + 1$. Unwind to get $TI(n) = n$
- also $TS(0) = 1$ and $TS(n) = TS(n-1) + TI(n-1)$. Unwinding we get the same thing as $TS(n) = n^2$
- However because it is the recursive method, we are calling the function repeatedly, which is actually more expensive. So the iterative implementation is actually more efficient than the recursive implementation.

Finally, $T(n) = O(n^2)$ in the worst case, but in the average case it is not $O(n^2)$. In the best case it can be very close to $O(n)$. So if the input is almost sorted, then insertion sort takes very less time which is not true for selection sort because there every input takes this much time.

In python the recursive implementation is going to throw an error without `sys.setrecursionlimit((2**31)-1)`

3.1.2 Merge sort

Both selection and insertion take too much time. Our next strategy would be to split the list into two halves, sort them independently and then merge them.

So our algorithm would be:

- Sort `A[:n//2]` and `A[n//2:]`
- Merge the two sorted halves to `B`
- How do we sort `A[:n//2]` an `A[n//2:]`? We do this recursively
- Merging lists:
  - if `A` is empty, copy `B` into `C`
  - if `B` is empty, copy `A` into `C`
  - otherwise compare the first elements of `A` and `B`. Move the smaller of the two to `C`
  - Repeat till all elements of `A` and `B` are moved.

3.1.3 Analysis of merge sort

First we analyse the merging procedure:

- Merge `A` of length m and `B` of length n
- In each iteration we add (at least) one element to `C`
- Hence merging procedure takes $O(m+n)$. But in our parts since we are splitting the array into roughly two halves, we should have $m \approx n$. Thus merging takes $O(2n) = O(n)$ time.

Then we analyse the merge sort procedure:

- Let $T(n)$ be the time taken of input of size $n = 2^k$ (for the sake of simplicity)
- recurrence:
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n = 4T(n/4) + 2n = 2^2T(n/2^2) + 2n$
  - ultimately we should have $T(n) = 2^kT(n/2^k) + kn = 2^{\log(n)}T(1) + n\log(n) = n + n\log(n)$

Ultimately the sorting procedure takes $O(n \log n)$ time, and hence can be used safely on large inputs. The negative aspect is the space requirement. Merge needs to create a new list to hold the merged elements. There is no obvious way to efficiently merge the lists in place. Extra storage can be costly.

4 October 20, 2022

4.1 Sorting algorithms cotd

4.1.1 Quick sort

Shortcomings of merge sort:

- merge needs to create na new list (space ineff)
- inherently recursive (hence more expensive space and time wise)
- merging happens because elements in the left half need to move to the right half and vice versa.

We want to do divide and conquer without merging. Suppose the median of `l` is m. We move all values of `l` atmost m to the left half of m. Then the right half is greater than m. Recursively sort both halves, and then there is no reason to merge, as we can just concat the sorted halves. The recurrence relation is $T(n) = 2T(n/2) + n$. Rearrangement can be done in a single time pass so $O(n)$. So $T(n) = O(n \log n)$.

The problem reduces to finding the median so as to ensure both halves have equal length. Naively we cant do so without sorting, but there is a method. Instead we pick any element called the pivot. We split `l` with the pivot element.

So we have the following sorting algo:

- choose the pivot element. Typically it is the first element in the array.
- partition `l` into lower and upper halves with respect to the pivot.
- Move the pivot between the lower and the upper parts. All the copies of the pivot should be in the lower part. The array should look like below:

$$\boxed{\cdots \leq p \,\|\, p \,\|\, > p \cdots}$$

- We want the rearrangement of the lists to be done in-place

Partitioning:

- scan the list from the left to right
- there are four segments: pivot, lower, upper and unclassified
- check the first unclassified element
  - if it is greater than the pivot, extend the upper to this element
  - if it atmost the pivot, then swap it with the first elment in upper. This extends lower and shifts upper up by 1.
- maintain two indices to mark the ends of lower and upper.
- exchange the pivot with the last element of lower.

What we achieved here is that the pivot is now in its correct position in the sorted array. We can do iteratitvely but then we dont do the rearrangement in place and hence we create lots of intermediate lists which is bad space consumption. This is not the only partitioning strategy, we can partition from the two extremes of the array as well.

4.1.2 Analysis of quick sort

- partitioning with respect to the pivot takes time $O(n)$. If the pivot is the median, then we have the complexity to be $O(n \log n)$
- Worst case is that at every step the pivot is the max or the min, and then the sizes of the partitions are 0 and $n-1$. Then we should have $T(n) = T(n-1) + n$. Hence we should get $T(n) = O(n^2)$. Even though this is the same as merge sort, it really doesnt give us the full picture.
- We can calculate that the average time complexity is $O(n \log n)$

What if we introduce randomization? Any fixed choice of pivot allows us to construct worst case input. Choose the pivot randomly at each stage.

4.1.3 Stable sorting

Often we are not sorting in isolation; we are sorting along one component where other components must maintain a relation among themselves. That is, if there is some existing order of other related attributes, then sorting another attribute shouldn't disturb this attribute. Quick sort as described is not at all stable, as swapping values while partitioning can disturb this order.

Often there are other criteria as well. We might want to reduce the effort of sorting repeatedly. Quick sort is the algorithm of choice generally, but merge sort is good for external sorting: database tables that are too large to store in memory all at once. Sometimes hybrid strategies are used too.

5 October 25, 2022

5.1 Stacks

- stakc is a LIFO squence
- `push(s, x)` → add `x` to `s`
- `pop(s)` → return the most recently added element
- maintain stack as list, implement push and pop as list operations from the right. Push will be append; and Pop is already a function as pop
- normally there is a third operation - `check_empty`

We have seen that we can encapsulate this in a class. The most common application would be function calls: when the function is called, the current frame of the function is pushed into the stack. When the function exits the frame is popped off.

5.2 Queue

- FIFO sequence
- the functions needed are `addq(q, x)` and `removeq(q)`
- We can use python lists, where we insert from the left and remove from the right.

An application would be *breadth first search*. Its generally maintained in scheduling tasks, where tasks are scheduled in a queue. Its a job scheduler.

5.3 Priority Queue

Sometimes a job scheduler may recieve a high priority job which needs to be prioritised over everything else. New jobs may join at any time, but when the processor is free then the job with the highest priority is executed first. So we need to maintain a collection of items with priorities to optimise the following operations: `delete_max()` →identify and remove item with highest priority and this need not be unique; `insert()`.

What if we keep an unsorted list? Insert is $O(1)$, but delete is $O(n)$. What if we keep a sorted list? Then the delete is $O(1)$, but insert is $O(n)$ (this is basically insertion sort). So in either case we kind of see that we are sacrificing one of them. So if we do this, in the worst case we are going to deal with $O(n^2)$.

Our argument being we can't do much with a 1-D data structure.

- assume N processes enter/leave the queue
- we store it in approximately $\sqrt{N} \times \sqrt{N}$ grid
- each row is sorted in the ascending order

- keep track of size of each row
- When we are inserting something we may insert in the first row which has an empty space, to do this we scan the size of the rows, and choose the first one which is not filled. Once we get the row, we do the usual insertion sort.
- Scanning size takes $O(\sqrt{N})$ and insertion takes $O(\sqrt{N})$. Insertion takes $O(\sqrt{N})$ then.
- maximum in each row is the last element. But this maxima is not sorted from top to bottom, but we know it is one of them. Since we know the size, it is basically the index of the last element. We can now look for the max in brute force, and then delete it and decrement the respective size. This takes $O(\sqrt{N})$ time.
- Thus processing N times takes $O(N\sqrt{N})$.

We can further optimise this using a binary tree.

5.4 Binary Tree

- values are stored as nodes in a rooted tree
- each node has upto two children
- other than the root, each node has a unique parent
- leaf node $\rightarrow$ no children
- size $\rightarrow$ number of nodes
- height $\rightarrow$ number of levels or longest distance from the root to a child

A special kind of binary tree is a heap.

5.5 Heap

- binary tree filled level by level, from left to right
- the value at each node is at least as big as te values of its children
  - max-heap
  - basically for every vertical path i follow, it will be sorted
- root always has the highest value
- holes are not allowed - every level must be completely filled from left to right

Inserting a value:

- we temporarily insert the new value as a leaf maintaining the heap structure
- we look up and see if it is bigger than its parent - if it is swap
- repeat until the above condition is not true
- we basically restored the heap property along the path to the root
- complexity?
  - we need to walk up from the leaf to the root
  - we are walking up only one path, so we at max traverse the height
  - number of nodes at level 0 is 2^0 nodes... at level j we have 2^j nodes

- if we fill k levels: we have $2^k - 1$ nodes. So if we have N levels, we will have $1 + \log(N)$ levels at most
- so we have $O(\log N)$ complexity

Deleting a value:

- we know that the root is the biggest value. So we can delete it easily
- after we delete root, we need to delete the rightmost element at the lowest level
- we move this node to the root
- then we swap this node with the biggest child (if it is smaller than both), and we continue
- complexity is same as above

Implementation:

- store as a list $H = [h_0, \ldots, h_n]$
- children of $H[i]$ are at $H[2^i + 1]$ and $H[2^i + 2]$
- parent of $H[i]$ is at $H[(i-1)//2]$ for $i > 0$

Better heapify

- list $l = [v_0 \ldots, v_n]$ is heap
- $mid = len(l)//2$. Since $l[mid :]$ has only leaf node
- cost turns out to be $O(n)$

5.5.1 Heap sort

- start with an unordered list
- build a heap - $O(n)$
- call `delete_max()` n times to extract elements in descending order $O(n \log n)$
- after each delete, size of heap shrinks by 1 (swap to the last element and dont touch it)
- store maximum value at the end of current heap
- in place $O(n \log n)$ sort

6 October 27, 2022

6.1 Search trees

Dynamic sorted data:

- sorting is useful for efficient Searching
- what if the data changes dynamically? (items are periodically deleted or added)
- insert/delete in a sorted list takes time $O(n)$
- So we move to a tree like structure - like heaps for priority queues
- our ultimate goal being reducing this to logarithmic time

6.1.1 Binary search tree

- this doesnt require any such structural constraint for now. It can be any binary tree but
- for each node with value v
  - all values in the left subtree are $< v$
  - all values in the right subtree are $> v$
  - this relation is recursive
- we assume that there are no duplicates

Implementation of binary search tree:

- each node has a value and pointers to its children. This is different from the very regualr structure that we saw in a heap
- add a frontier with empty nodes, all fields empty
  - empty tree is single empty node
  - leaf node points to empty nodes
  - kind of like how we worked for empty linked list
- three local fields `value, left, right`
- value `None` for empty value
- empty nodes have all fields `None`
- leaf has a nonempty `value` and empty `left` and `right`
- Inorder traversal:
  - list the left subtree, then the current node, then the right subtree
  - lists values in sorted order
  - use to print the tree recursively
  - sanity check to see the that the tree that we have constructed is actually ok

Finding a value v:

- check value at current node
- if v smaller than current node, go left, else right
- this is like a natural generalisation of binary search

Maximum and minimum:

- minimum is the left most node in the tree (it has no left, but may have a right)
- maximum is the right most node in the tree (it has no right, but may have a left)

Inserting a value v:

- try to find v - if we find it that means it exists so leave it be
- insert at the position where find fails - basically replace the empty node with the value to be inserted

Deleting a value v:

7 November 1, 2022

7.1 Operations on search trees

- `find()`, `insert()`, `delete()` all walk down a single path
- worst case: height of the tree
- an unbalanced tree with n nodes m ay have height $O(n)$
- Balanced trees have height $O(\log(n))$
- How can we maintain balance as the tree grows and shrinks?

Defining balance:

- left an dright subtrees should be equal - two possible measures - **height** and **size**
- `self.left.size()` and `self.right.size()` are equal? - only possible for binary search trees
- `self.left.size()` and `self.right.size()` only differ by 1? - plausible but difficult to maintain
- `self.height()` - umber of nodes on longest path form root to leaf
  - 0 for empty tree
  - 1 for tree with only a root node
  - $1 + max$ of heights of left and right subtrees in general
- height balance - `self.left.height()` and `self.right.height()` only differ by atmost 1 (AVL *trees*)
- Does height balance guarantee $O(\log(n))$ height?

We sort of try to see which is the smallest tree (by size) that we can construct for a given height and see if we can find out some pattern and then we try to generalise it. Generally, $size \propto 2^{height}$.

- General strategy to build a small balanced tree of height h
  - smallest balanced tree of height $h-1$ as left subtree
  - smallest balanced tree of height $h-2$ as right subtree
- $S(h)$ - size of smalles height balanced tree of ht h
- recurrence:
  - $S(0) = 0, S(1) = 1$
  - $S(h) = 1 + S(h-1) + S(h-2)$
- this is almost similar to the fibonacci sequence
- $S(h)$ grows exponentially with h
- for size n, h is $O(\log(n))$

Correcting imbalance:

- Slope of a node: `self.left.height()` - `self.right.height()`

- we want this to be in $\{-1, 0, 1\}$
- insert or delete can alter slope to -2 or $+2$
- we can rotate the trees
  - right rotation - slope $+2$ to $\{-2, -1, 0\}$
  - left rotation - slope -2 to $\{0, +1, +2\}$

7.2 Memoization

7.2.1 Inductive definitions, recursive programs, subproblems

- factorial:
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$
- insertion sort: (subproblem = sort a smaller list)
  - $isort([]) = []$
  - $isort([x_0, x_1, \ldots, x_n]) = insert(isort([x_0, \ldots, x_{n-1}]), x_n)$

We can define the function as simple recursive funtions. In many cases we inductively solve subproblems which are of the same type. Solution to original problem can be derived by combining solutions to subproblems. What causes trouble is when the subproblems interact. Meaning, we come to solve subproblems which we have already solved, but since we havent stored the result, we will need to solve it again, which is wasteful. For example fibonacci. A simple recursive solution is okayish, but remember that we are actually calling $fib(k)$ for some $k \leq n$ multiple times. computation tree grows exponentially. A lookup table of some sort is a better solution.

Evaluating subproblems:

- build a table of values already - like a memory table
- **Memoization** - check if the value to be computed has already been seen before
- look up the table before making a recursive call, and then computation tree becomes linear

General angle of attack

def $f(x, y, z)$:
    if (x, y, z) in $ftable.keys()$:
        return $ftable[(x, y, z)]$
    recursively compute from subproblems
    $ftable[(x, y, z)] = value$
    return $value$

8 November 3, 2022

8.1 Dynamic programming

- anticipate the structure of subproblems
  - derive from inductive definition
  - dependencies area acyclic
- solve subproblems in appropriate order
  - start with base cases - no dependencies
  - evaluate a value after all its dependencies are available
  - fill it iteratitvely without ever making a recursive call. Doesnt mean we are bypassing the inductive definition; dynamic programming is a more efficient way of evaluating the inductive definition

8.1.1 Grid paths

- suppose we have a rectangular grid of $m \times n$
- we can only go up or right from $(0, 0)$ to (m, n)
- then how many paths are there? $\binom{m+n}{m}$
- What if a certain point in the grid is blocked. Assuming that we cant pass through that point, how many ways are there? Easy combinatorial solution exists using the previous logic.
- What if now two points are blocked? Same logic as the previous one, inclusion exclusion principle.

Inductive formulation of the above problem

- How can a path reach (i, j)?
  - move up from $(i, j-1)$
  - move right from $(i-1, j)$
  - note that in both cases the paths are disjoint, there is no path which can come from both directions, and any path from these two neighbours can reach the destination in only one unique way.
- recurrence for $P(i, j) \to$ number of paths:
  - $P(i, j) = P(i-1, j) + P(i, j-1)$
  - $P(0, 0) = 1$ (base case), it might be tempting to say 0, but there is 1 way to stand at $(0, 0)$ and not move
  - $P(i, 0) = P(i-1, 0)$ - bottom row
  - $P(0, j) = P(0, j-1)$ - left column
- $P(i, j) = 0$ if theres a hole at (i, j)

So how do we do it? $P(0, 0)$ has no dependencies. We start at $(0, 0)$, and then we fill row by row. We forcibly set every hole to 0, and keep using the inductive formula.

We could have filled column by column, and that wouldnt have made any difference. We can also do diagonal wise, in which case we get a pascal's triangle.

Memoization computes only those values which will be needed. Dynamic programming will store all those values which may be needed.

8.1.2 Longest common subword

Our goal is to find the length of the longest common subword between two strings. Formally speaking:

- $u = a_0a_1 \dots a_{m-1}, v = b_0b_1 \dots b_{n-1}$
- common subword of length k - for some positions i and j: $a_ia_{i+1} \dots a_{i+k-1} = b_jb_{j+1} \dots b_{j+k-1}$

We can start by comparing the first letters of the word; if they are same, then count $+1$ and then look to the right of both. If they dont, then we have to do something different. So we inductively formulate this first:

- Find the largest k such that for some positions i and j
- $LCW(i, j)$ - length of common subword in $a_i \dots a_{m-1}$ and $b_j, \dots b_{n-1}$
  - if $a_i \neq b_j, LCW(i, j) = 0$
  - if $a_i = b_j, LCW(i, j) = 1 + LCW(i + 1, j + 1)$
  - base case: $LCW(m, n) = 0$
  - in general, $LCW(i, n) = 0 \forall 0 \leq i \leq m$ and $LCW(m, j) = 0 \forall 0 \leq j \leq n$

Subproblems:

- subproblems are of the form $LCW(i, j)$ for $0 \leq i \leq m, 0 \leq j \leq n$
- we should have a table of $(m + 1) \cdot (n + 1)$ values, with the top left corner where the strings start
- $LCW(i, j)$ depends on $LCW(i + 1, j + 1)$. So the direction of attack is diagonally from the bottom right to the top left
- because of the base case, the bottommost row and the rightmost column are all 0's
- we can start at bottom right, and then as the previous problem, fill row by row or column by column

How do we read the solution? We first find the entry (i, j) with the largest LCW value. Then from there on we keep continuing diagonally till we have non zero values. The brute force algorithm was $O(mn^2)$ but here we have $O(mn)$.

8.1.3 Longest common subsequence

This is same like the previous one, but this time we are allowed to drop letters in between: we want the longest common substring of letters where letters in that substring may not be consecutive. The answer is easy once we know the previous one: it is the longest path

of non zero entries in the 2-D table constructed above, where we can move only down and right from the top left. But this is difficult to implement.

We apply this in analyzing genes (which is a long string over A, T, G, C). The `diff` command in Unix compares text files and finds the longest matching subsquence of lines. Each line of text is a character.

Imagine this as there being two horizontal lines, of the two strings. Ths same letters in both the strings are connected with a line. If we take the longest common subsquence, then no connecting lines intersect each other. This is because the order of the letters in the longest common subsquence needs to stay fixed.

What is the inductive structure?

- $LCS(i, j)$ - length of common subsquence in $a_i \dots a_{m-1}$ and $b_j, \dots b_{n-1}$
- if $a_i = b_j$, $LCS(i, j) = 1 + LCS(i + 1, j + 1)$. We can assume that (a_i, b_j) is a part of the LCS
- if $a_i \neq b_j$, then a_i and b_j cannot be both part of the LCS as then the connecting lines would have to intersect. So which one should we drop? Solve $LCS(i, j + 1)$ and $LCS(i + 1, j)$ and take the max
- base cases are same as LCW

The length is at $(0, 0)$. Trace back the path by which each entry was filled. Each place where a diagonal edge starts, it is part of our solution.