Analysis of Algorithms

Roudranil Das

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

October 20, 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:

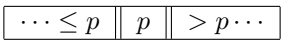
- merge needs to create na new list (space ineff)
- inherently recursive (hence more expensive space and time wise)
- merging happens because elements in the left half need to move to the right half and vice versa.

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:



- 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.