Data Structures and Algorithms

Short notes

Roudranil Das

May 24, 2022

Contents

Algorithms

1 Introduction

Data structures and algorithms play a very important role.

Some tips learnt:

- Use stress testing.
- Stress testing is done by randomly generating vaild inputs, for large parameter values. Once error is obtained, start decreasing values to get a small case.

2 Some short examples

2.1 Fibonacci numbers

One of the most easiest ways to return the n^{th} Fibonacci number is recursive method. However this algorithm is extremely slow, mainly because while computing F_n we compute each F_{n-r} multiple times.

One way to get around this is to write down all Fib numbers, a list like method, as shown below:

```
        F = []
        F[0] = 0
        F[1] = 1

        for i from 2 to n:
            F[i] = F[i-1] + F[i-2]

        return F[n]

```

The recursive one would need atleast F_n many iterations to compute F_n which would grow tremendously with valeus of n. The second one will not.

2.2 GCD

The naive way would be to run d from 1 to $a + b$, check if $d \mid a$ and $d \mid b$ and return the largest d. This is obviously slow. The faster way is to implement Euclidean algorithm recursively. The total number of steps required is $\approx Log(ab)$.

3 Computing runtimes

Runtime analysis of the Fibonacci code:

```
        F = [] # memory management
        F[0] = 0 # assignment
        F[1] = 1 # assignment

        for i from 2 to n: # increment, comparison, branch (condition)
            F[i] = F[i-1] + F[i-2] # lookup, assignment, addition of large integers

        return F[n] # lookup, return from stack

```

So each line of code takes different amount of time. We want a new way to measure the runtime without knowing a large amount of metadata. For this we move on to asymptotic notation.

3.1 Asymptotic notation

All of these factors, usually although they change runtimes, they change it by a constant factor. So we measure runtime by ignoring these constants. We actually look at asymptotic runtime, where we measure how the runtime scales with the input size n. We see how it is proportional to some function of n. The order being:

$$\log(n) < \sqrt{n} < n < n\log(n) < n^2 < 2^n$$

3.2 Big-O notation

Definition: $f(n)$ is called *Big-O* of $g(n)$, i.e.,$f(n) = O(g(n))$ $(f \preceq g)$, if $\exists N \in \mathbb{N}$ and $c \in \mathbb{R}$ such that $\forall n \geq N, f(n) \leq c \cdot g(n)$.

f is bounded above by some multiple of g. Also base of logarithm doesnt matter as any base is essentially a constant multiple.

The disadvantage is that the constant multiple info is lost. So first improve the asymptotic situation. Then look into the smaller details. Also it may be the case that even if algo A is better in Big-O than algo B, for all practical purposes, algo B is better.

Some common rules:

1. Multiplicative contsants can be omitted. For example:
   $7n^3 = \mathrm{O}(n^3)$, $\frac{n^2}{3} = \mathrm{O}(n^2)$.
2. $n^a < n^b$ for $0 < a < b$:
   $n = \mathrm{O}(n^2)$, $\sqrt{n} = \mathrm{O}(n)$.
3. $n^a < b^n$ $(a > 0, b > 1)$:
   $n^5 = \mathrm{O}(\sqrt{2}^n)$, $n^{100} = \mathrm{O}(1.1^n)$
4. $(\log n)^a < n^b$ $(a, b > 0)$:
   $(\log n)^3 = \mathrm{O}(\sqrt{n})$, $n \log n = \mathrm{O}(n^2)$
5. If there is a sum of terms smaller terms in the sum can be omitted:
   $n^2 + n = \mathrm{O}(n^2)$, $2^n + n^9 = \mathrm{O}(2^n)$

Next we demonstrate Big-O in practice by analysing the runtime of the Fibonacci code:

Operation	Runtime
Create an array $F[0, 1 \ldots n]$	$\mathrm{O}(n)$
$F[0] \leftarrow 0$	$\mathrm{O}(1)$
$F[1] \leftarrow 1$	$\mathrm{O}(1)$
for i from 2 to n:	loop $\mathrm{O}(n)$ times
$F[i] \leftarrow F[i-1] + F[i-2]$	As integers are vvery large, $\mathrm{O}(n)$
return $F[n]$	$\mathrm{O}(1)$

Total: $\mathrm{O}(n) + \mathrm{O}(1) + \mathrm{O}(1) + \mathrm{O}(n) \cdot \mathrm{O}(n) + \mathrm{O}(1) = \mathrm{O}(n^2)$

There are some other notations too: Ω is used to give an approximate lower bound of the runtime, and Θ is used to give an approximately equal runtime.

Little-o notation o(): $f(n) = \mathrm{o}(g(n))$ or $f \prec g$ if $\frac{f(n)}{g(n)} \to 0$ as $n \to \infty$.

4 Greedy Algorithms

Greedy algorthms solve for local optimums in order to reach a global optimum. It involves breaking up a problem into a chunk of smaller problems, find a starting optimal solution, which will leave us with a smaller subproblem for which another optimal solution can be found by the same algorithm. Then we combine all the optimal solutions.

Main steps followed are:

- Make a first move
- Solve a problem of the same kind
- Then solve a subproblem of the same kind

A move is called a *safe move* if there is a optimal solution consistent with this move. Not all greedy moves or all first moves are safe.

Example: n children come to a party. Find the minimum number of groups to split the children into so that in any group the age of the children differ by at most 1.

Solution: A naive approach would be to consider all possible partitions of the set of all children and iterate over them to find the suitable partition, but it's complexity is $\Omega(2^n)$ which is not viable at all. So before anything we fomulate the problem mathematically. □

Covering points by segments

Input: A set of points $x_1, x_2, x_3, \ldots, x_n \in \mathbb{R}$
Output: The minimum number of segments of unit length needed to cover all the points.

Safe move: Cover the left most point with a segment whose left end is at that point. Then start with the first point left to the right of the segment, and continue with the same strategy.

Pseudocode: (assuming points are sorted in ascending)

PointsCoverSorted$(x_1, x_2, \ldots, x_n)$

$R \leftarrow \{\}, i \leftarrow 1$
while $i \leq n$:
    $[l, r] \leftarrow [x_i, x_i + 1]$
    $R \leftarrow R \cup \{[l, r]\}$
    $i \leftarrow i + 1$
    # Basically find the next point to the right of the current line segment
    while $i \leq n$ and $x_i \leq r$:
        $i \leftarrow i + 1$
return R

The time complexity is $O(n)$. In case points are not sorted, sort them by an algorithm of complexity $O(n \log(n))$. Thus final complexity will be $O(n \log(n))$.

4.1 Fractional Knapsack

Formulating in mathematical terms:

Fractional Knapsack

Input: Weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ of n items; capacity W.
Output: The maximum total value of fractions of items that fit into a bag of capacity W

We state the following lemma of the safe move:

Lemma

There exists an optimal solution that uses as much as possible of an item with the maximum value/unit of weight.

Greedy algorithm:

- while Knapsack is not full
- Choose item i with maximum $\frac{v_i}{w_i}$
- If item fits into the knapsack completely, then take all of it
- Otherwise take as much as to fll the knapsack entirely
- Return total value and amounts taken

The pseudocde is given as below:

Knapsack$(W, w_1, v_1, \ldots, w_n, v_n)$

$A \leftarrow [0, 0, \ldots 0], V \leftarrow 0$
repeat n times:
    if $W == 0$:
        return (V, A)
    select i with $w_i > 0$ and max $\frac{v_i}{w_i}$
    $a \leftarrow min(w_i, W)$
    $V \leftarrow V + a\frac{v_i}{w_i}$
    $w_i \leftarrow w_i - a, W \leftarrow W - a, A[i] \leftarrow A[i] - a$
return (V, A)

Complexit is O(n^2): why? Because selecting best item takes O(n) and loop is executed n times. **It is possible to optimise!! How?**

- Sort the array of $\frac{v}{w}$ in descending order
- In the above case we can skip the select statement
- after accounting for the sorting algorithm, we can sort in O($n \log n$)