Algorithms - 1 Brief Introduction

What we focus on the algorithms:

1. Correctness:

> In order to make sure that algorithm works, we need to
>
> > a. prove result is correct
> > b. prove that the algorithm always terminate
>
> Normally we can use Loop Invariant to prove the correctness:
>
> > **Loop invariant:** A property that is true before the start of each loop, and true just after the loop ends
>
> > Initialization: similar to base case - Prove the loop invariant is true prior to the first iteration of the loop
> > Maintenance: If the loop invariant is true before an iteration, it remains true before the next iteration
> > Termination: When the loop terminates, the loop invariant gives us a useful property that hepls show that the algorithm is correct
> >
> > Examples:
> >
> > > **insertion sort**
> > >
> > > ```
> > > INSERTION-SORT(A)
> > >     for j in range(2, len(A)): # after every loop A[:j] is sorted
> > >         key = A[j]
> > >         i = j - 1
> > >         while i> 0 and A[i] > key:# find a suitable place for key
> > >         A[i+1] = A[i]
> > >         i = i-1
> > >             A[i+1] = key
> > > ```
> > >
> > > *Outer Loop invariant:* At the start of each iteration of the for loop index by j, the subarray A[1..j-1] consists of the elements originally in A[1...j-1], but in sorted order

Inner Loop Invariant: At the start of each iteration of the while loop, the subarray A[i+1...j] consists of elements greater than the key, and contains the items originally in A[i+1..j-1] in the same sorted order. The items in position A[1..i] are unchanged.

pow

https://stackoverflow.com/questions/39492263/prove-correctness-of-power-algorithm-using-loop-invariance

```
def Pow(x,y)
    e = 1
    while(y>0):
        if y%2 ==0:
         x = x*x
           y = y/2
        else:
            e = e*x
            y = y - 1
  return e
```

y_i, e_i, x_i for i^{th} iteration of the loop
loop invariant: p_i>= 0 and $e_i(x_i)^{y_i} = (x_0)^{y_0}$
loop terminates, there exists iteration n such that $y_n = 0, e_n = (x_0)^{p_0}$

2. Running Time

How can we estimate the running time based on the size of the input: **RAM**

each of normal instructions takes a constant amount of time
running time = $\sum cost\ of\ statement\ \cdot\ number\ of\ itmes\ statement\ is\ executed$

Usually we use worst case, when cover randomized algorithms use average time

Asymptotic Notation

used to describe the growth of function whose domain is the set of natural numbers
drop lower-order terms, drop constant coefficients

Asymptotic Upper Bound Notation:
Big-O: $f(n)$ is $O(g(n))$ if there **exists** a $c > 0$ and an n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$
Little-o: $f(n)$ is $o(g(n))$ if for **any** $c > 0$ and an n_0 such that $f(n) < cg(n)$ for all $n \geq n_0$

Asymptotic Lower Bound Notation:
Big-Omega: $f(n)$ is $\Omega(g(n))$ if there **exists** a $c > 0$ and an n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$
Little-o: $f(n)$ is $\omega(g(n))$ if for **any** $c > 0$ and an n_0 such that $f(n) > cg(n)$ for all $n \geq n_0$

Asymptotic Lower Bound Notation:
Big-Theta: $f(n)$ is $\theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$

Some useful facts:

$log_{200} n \in log_2 n$
$logn! \in nlogn$
n^n dominant $n!$ dominant C^n

$log_a b = log_a c^{log_c b} = log_c b \cdot log_a c$
Stirling's approximation: $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + O(\frac{1}{n}))$

https://www.desmos.com/calculator/0zirxhft0q Drawing lines

Structure of this folder:

datas tructure

> heap, hashtable, tree,

Algorithms

divide and conquer

> matrix multiplication, big integer multiplication, closest pair points

Sorting

> insertion sort, heapsort, quick sort, ...

Reference/ Good resources

Algorithm Design and Applications[A4]
CLRS