|  |  |  |
| --- | --- | --- |
| Power | Val | Cs speak |
| 0 | 1 |  |
| 1 | 2 |  |
| 2 | 4 |  |
| 3 | 8 |  |
| 4 | 16 |  |
| 5 | 32 |  |
| 6 | 64 |  |
| 7 | 128 |  |
| 8 | 256 |  |
| 9 | 512 |  |
| 10 | 1k | Kilo |
| 20 | 1m | Mega |
| 30 | 1b | Giga |
| 40 | 1t | Tera |
| 50 | 1q | peta |
| 60 | 1z | Exabyte (google had 10 in 2013) |

= num ways to pick k from n (**no order**)

= choose k from n **with order**

n! = permutations

= 1 + 2 + … + n

**BAYES RULE:**

½ + ¼ + … = 1

sigma(x) = sigma’(x) = s(x) ( 1 – s(x) )

(on when y=1)y (on when y=0)1-y

**# nodes** **in tree** with depth D and branching factor b

1 + b + b2 + … + bD

= bD+1-1/b-1

**Processes** have their own address space

**Threads** are in a process, share heap but separate stacks

Numerator **OR** denominator can be float

**BIT OPS**

-reads right-left

-$

& = **and**

^ = **xor**

| = **or**

~ = **complement**

int(log(x, 2)) + 1 = **LENGTH**

**XOR**

If 0, flow through

If 1, bit flipper

X^X = 0

Swapper:

X = x^y

Y = y^x

X = x^x

**HEAPS**

Parent = (I – 2) / 2

i

left = i\*2 right = i\*2+1

import heapq

heapq.heappush(list, item)

heapq.heappop(list, item)

heapq.heapify(list)

heapq.pushpop(list, item)

**SEARCH**

import bisect

bisect.bisect\_left(a, x, low=0, high=len(a))

index of leftmost val >= x

**QUEUES**

From collections import deque

D =deque([1, 2, 3])

d.appendleft(x)

d.extendleft([…])

d.pop()

**REGEX**

|  |  |
| --- | --- |
| \d | [0-9] |
| \D | [^0-9] |
| \s | [whitespaces] |
| \S | [^whitespace] |
| \w | Alphanumeric |
| \W | [^alphanumeric] |
| \* | Repeat 0+ |
| + | Repeat 1+ |
| {m, n}  {m,}  {,n} | n-n times |
| ? | 0 or 1 |
| | | Or |
| ^…$ | Start and end of line |
| \b | Word boundary |
| (…) | group |

Import re

Re.search(pattern, string)

Tmp = re.compile(pattern, string)

Tmp.search(string)

Re.findall(pattern, string)

**(Returns STRINGS, not re.match)**

Re.finditer(pattern, string)

Re.sub(pattern, string, replace)

Match.group(group #)

Match.start()

|  |
| --- |
| Match.end() |
| FLOAT\_PATTERN="^\-?\d\*\.\d+$" |
| INT\_PATTERN = "^\-?\d+$" |  |
| SCI\_PATTERN="^\-?(\d\*\.)?\d+e\+\d+$" |  |
|  |  |

**TREE TRAVERSAL**

Tree depth **D**

Soln is at depth **d**

Branching factor **b**

|  |  |  |
| --- | --- | --- |
|  | Space | Time |
| DFS | O(D) | O(bD) |
| BFS | bd | bd |
| DFS + iterative deepening | O(d) | O(bd) |

* If bisect\_left can’t find x, it returns the index of NEXT LARGEST element
* Isinstance(x, (dict, int))
* Check if arrays are sorted!!
* Math.log10
* Raise **StopIteration**
* Permutation invariant hash = summing hashes of each element!
* Consider extreme base cases!!
* XOR is a fundamental operation!!!
  + Xor = AB’ + A’B
* Examples are your best friend vs tricky indexing!!
* Duplicating/adding arrays to itself is one of your fundamental ops!!
* **Pairs of a:** [a[i:i+2] for I in range(len(a)-1)]
* **Getattr(node, ‘next’, None)**
* **be careful!!!!!!!!!** Consider all cases and read problem statement
* consider pre/post/in traversal!
* Recursive gnerators are fun ☺
* Best way to check for neighbors
  + **Make OFFSETS constant**
  + Yield each offset if application is valid
* Sorted() goes to **<**
* Think about problems in terms of smallest/base case!!
  + E.g. random deck, deck of 1, then expand
* Prime carry bit w/1 for sums, list with dummy head for traversals, etc
* Wait to implement little optimizations until you have something working
* Sys.maxint
* Ask if can define own linked lists, etc (e.g. to get length)
* **Ival overlap**: max(starts) < min(ends)
* Do things by hand!!
* Ask if need to do in place
* Be **CAREFUL!! ESPECIALLY ABOUT INDEXING**
* When using generators, always worry about StopIteration
* Max/min is fundamental seq op
* Copy lists w/ [:] for recursive calls
* Find missing: sum(1 .. n) - sum(A)
* All caps constants are your friend ☺
* Throw away cases that are tripping you up, tackle later
* Consider bit vectors :0
* Recover forward idx from reversed iteration: **len(a) – i – 1**
* Tuples are immutable
* Assume part of problem is done, how would you expand?
* Consider hanging cases! (when past loop but still have stuff to check for)
* From itertools import **product**
  + For (x, y) in product(range(rows), range(cols):