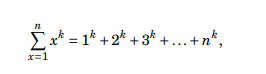
**Competitive Programming**

1. A.P: Arithmetic Progression, to get the sum of numbers where the diff is constant.





and for xk



There’s a general formula for sums, Faulhaber’s formula.

General formula for AP series:



* 1. Sum of all squares:

S= n(n+1)(2n+1)/6

1. G.P: A geometric progression is a series when ratio between any 2 consecutive numbers is constant.





1. Harmonic Sum: …
2. Set Theory:
3. Log:

Normal exponents like 24 = 16 can be seen as, we move by 2 steps for 4 times and then we reach a distance of 16 steps. 20=1 means we didn’t move at all so we reach a distance of 1 step which is where we were already. 2-1=1/2 meaning we went 2 steps back from our current position. Log is just a way to denote where we ended up and what the size of our steps was. So 24=16 is written as log216=2.

* 1. Log properties:

logab = log a + log b

loga/b= log a – log b

log ab= bloga

alogcb= blogca

lognm = logpm / logpn

lognm = 1/ logmn

an=b is

logab = n

* 1. Similarly, log properties apply to exponentials too.

ab/ac = ab-c

ab\*ac = ab+c

1. function:
2. logic:
3. Criterias for analysis of algs: There are many but 2 most important, the time and the space complexity. We don’t care about the exact time or memory usage but the growth of it, so how fast does the time and the memory consumption grow in relation a given metric, usually the input size, denoted by n.
   1. Calculating Time&Space Complexity: We assume each statement takes 1 unit of time. Every variable is said to take 1 word of space.

For a pseudocode

for(int i{0}; i<n; i++) ---- n+1 times

{

//something --- n times

}

TC= 2n+1 times

Since we only need the complexity, degree of the polynomial is taken, which is just n in this case.

SC= i (i=1)

But since i only takes 1 element each time, it is counted as 1 word.

SC is 1 as it doesn’t grow with n.

For each value of i (0,1,2,3….) the inner statement will execute once for n times. Same for the for loop to check the value, except it will run 1 additional time to check the last value.

If there are multiple loops then we just add the TC for everyone and the degree is taken like normal.

* 1. For nested loops,

for (i=0; i<n;i++ ) --- n+1

{

for(j-0; j<n; j++) ---n \* (n+1)

{

//… ---n \* n

}

}

TC= 2n2+2n+1

Degree= n2

SC = 2n2+2

If j < i instead,

then

for (i=0; i<n;i++ ) --- n+1

{

for(j-0; j<i; j++) ---n \* (i+1)

{

//… ---i(i+1)/2

}

}

Since i isn’t too little than n we can say i=n

TC= n+1+ n2+n + (n2+n)/2

Degree= n2

as the inner loop runs for 0+1+2+3…+i times, and the difference is constant, we can use the Sum of AP formula to say it runs for i(i+1)/2 times

For non-n dependent loops,

p=0

for(i=1; p<=n; i++)

{

p=p+i;

}

lets say i executes k times max, then p will go for 0 + 1 + 2 + 3... k times growing with i.

i=1

p=0+1

i=2

p=1+2 (or 0 + 1 + 2)

i=3

p=3 +3 (or 0 + 1 + 2 + 3)

and so on till

i=k

p= 0+1+2+…+k

So the loop will stop when

p>n

therefore

k(k+1)/2 > n

which can be written as

k2>n

k > root(n)

Hence TC = root(n)

For growing increments,

for(i=0; i<n; i\*=2)

{

…

}

Since i grows with each value the increment is not constant,

i=1 then next is 1\*2= 2

i=2 then next is 2\*2= 22

i=4 then next is 4\*2 = 23

and so on

until 2k

exit condition is 2k = n

which is k=log2n

TC= log2n

For log in TC we take the ceiling for fractions so logn = 3.2 then we take it as 4.

Growing or decreasing are the same thing in TC,

so

for(i=n;i>=1;i/=2)

{

//

}

n/2 then n/2/2 which is n/22 then n/22/2 and so on

until

n/2k < 1

n < 2k

log2n < k

TC is log2n

for(i=1;i<=n;i++) --n

{

for(j=1; j<n; j\*=2) --n \* log2n

{

… --n \* log2n

}

}

TC= 2nlog2n+n

Here degree is nlog2n which is the TC.

* 1. For non-nested loops:

p=0

for (i=1; i<n; i\*=2)

{

P++;

}

TC = log2n

for (j=1l j<p; j\*=2)

{

…

}

TC= log2p

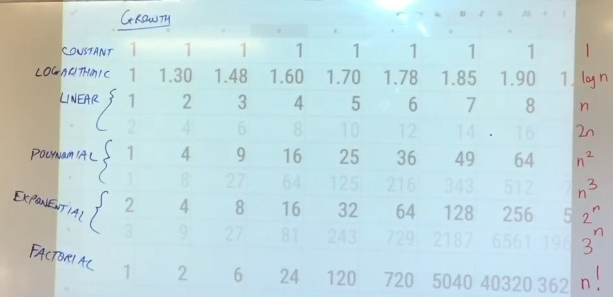
Since p++ is ran for log2n times Total TC = log2log2n

* 1. Conditionals: In case of conditional statements we just get the worst case complexity.
  2. TC Classes of Functions: We can order time complexity by the order of the functions,

In ascending order:

1 < logn < n1/2 < n < nlogn < n2 < n3 < …. < 2n < nn

Which can also be seen by



as the growth increases with each type of series.

* 1. Asymptotic Notations: Main are (theta, average bound), (big-omega, upper bound) and O (big-o, lower bound).
     1. big-O:

f(n)= O(g(n)) iff there exist +ve constant c and n0 such that f(n) <= c\* g(n) n > n0

: for all

which basically means,

if f(n)= 2n+3

then

2n+3 <= <a constant> \* g(n)

g(n) can be n, n2and any other degree of n.

so if

g(n)= n and c=10

2n+3 <= 10n

Then the starting value for which this eqn is satisfied is 1, this is the n0. So n > n0 will be written as n>1 for this function.

which means O(n) is the upper bound.

But, similarly,

g(n) can be n2 as well so O(n2) is also the upper bound. g(n) is any order after n including n, this is the upper bound.

We take O(g(n)) where the upper bound is the lowest amongst all other upper bounds. So O(n) in this case.

* + 1. big-omega:

f(n)= (g(n)) iff there exist +ve constant c and n0 such that f(n) >= c\* g(n) ∀ n > n0

Same as O but the only difference is the >=, which gives us (n), (n1/2) and so on for f(n)=2n+3. We take the highest lower bound which is n in this case.

* + 1. Theta: Average bound, recommended to give this than big-o. As big-O or the upper bound can be any value higher than the required value and can be any value lower than the required value, but the average value is always exact. However, Theta isn’t always possible so we give the big-O.

f(n)= (g(n)) iff there exist +ve constant c1,c2 and n0 such that c1\*g(n) <= f(n) <= c2\* g(n)

for f(n)= 2n+3

1n <= 2n+3 <= 10n

g(n) is n and therefore, (n) is the average bound. Unlike Upper and Lower bounds, there is only 1 average bound as any other value of g(n) voids the condition.

* + 1. Generally the time complexity is the same as the f(n), this is given through the Reflexive property of Asymptotic Notations which says,

if f(n) is given then f(n) = O(f(n)) and (f(n))

A little less generalization is,

Say,

f(n)= n!

n\*(n-1)\*(n-2)…\*3\*2\*1

which can be written as

1\*2\*3…\*n

We take the first value for c1\*g(n) and last value for c2\*g(n)

1\*1\*1 <= 1\*2\*3…\*n <= n\*n\*n\*n

So,

1 <= n! <= nn

Since we don’t have the same g(n) on both sides we can’t get the for n!, only O(nn) and (1).

We can also do it as,

½\*n! <= n! <= 2 \* n!

which gives g(n)= n!

However, since we don’t have a time function for n! we neglect it.

This generalization gives us bounds for all f(n).

For ex.:

f(n) = logn!

log(1\*2\*3\*…\*n)

Which according to the generalization becomes,

log(1\*1\*1\*1…) <= log(n!) <= log(n\*n\*n…)

1log1 <=log(n!) <= nlogn (because logxy = ylogx)

0 <= log(n!) <= nlogn

So, O(nlogn) and (1) (we assume constants as 1)

* + 1. General Property: if f(n) is O(g(n)) or (g(n)) or (g(n)) then a\*f(n) is also O/omega/theta(g(n))
    2. Transitive property: if f(n) = O/omega/theta(g(n)) and g(n)= O/omega/theta(h(n)) then f(n) = O/omega/theta(h(n))
    3. Symmetric Property: if f(n) is theta(g(n)) then g(n) is theta(f(n)), only true for theta.

For ex.

f(n)= n2 and g(n)=n2

then

f(n)= theta(n2)

g(n)= theta(n2)

* + 1. Transpose Symmetric: Only true for omega and big-o,

if f(n)=O(g(n)) then g(n) = omega(f(n))

which is,

f(n) is upper bound for g(n) and g(n) is lower bound for f(n)

For ex.

f(n)= n and g(n) = n2

then

n is O(n2) and n2 is omega(n)

* + 1. If f(n)= O(g(n)) and f(n)= omega(g(n)) i.e., c1=c2 in c1g(n) <= f(n) <= c2g(n)

then f(n)= theta(g(n))

* + 1. Addition: When 2 upper bounds are added then only the max of the 2 is taken,

i.e.,

f(n)= O(g(n))

d(n)= O(e(n))

let f(n)=n and d(n)= n2

then f(n)+d(n) or O(g(n))+ O(e(n)) = n2+n whose degree is n2, so

f(n)+d(n)= O(max(g(n),e(n)))

Inversely for omega it will be min.

* + 1. Comparison of time complexities:

First method is to just plot the values and directly observe.

Second is to apply log on both values

n2 vs n3

then log(n2) vs log(n3)

2logn vs 3logn

and 2logn < 3logn

* 1. Average, worst and best case:

Best case: The least no. of operations.

Worst Case: The max no. of operations

Average Case: (All possible case time)/no. of cases

For ex.

For linearly searching an array

best case = O(1)

Worst case = O(n)

Avg. case = (1+2+3+…+n)/n

which is

(n(n+1)/2)/n

n(n+1)/2n

(n+1)/2

Avg case= theta(n)

For each case, we can have upper bound, lower bound and avg. bound, so

Best case= O(1)

= omega(1)

= theta(1)

and similarly we can get other case bounds too.

Upper bound or O is not the same as the worst case.

* 1. Time complexity brief:

1: Constant time, no matter input size, output will take same time.

log2n: meaning the alg. halves the inp size at each step.

n1/2

n: linear time, for big inputs this is usually the most efficient alg as this means 1 loop for input, 1 for processing and 1 loop for output and all go n times.

nlogn: Indicates sorting alg, as the alg takes log n time on each element.

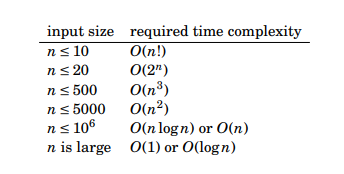
nk: k loops

2n: Indicates alg iterates through all subsets of the input set.

n!: Indicates alg iterates through all permutations of the input.

An alg is polynomial if at most it takes O(nk) time.

* 1. Inp Size and Required time complexity of the alg:



1. Sort: We use sort(begin iterator, end iterator); For user-defined types we overload ‘<’ operator. We can provide a bool function as predicate or as address to create custom comparator.

struct P {

int x, y;

bool operator<(const P &p) {

if (x != p.x) return x < p.x;

else return y < p.y;

}

};

1. Searching: To find an element we can iterate through the array, O(n) TC but if the array is sorted, using Binary search we can do it in O(logn) time.

Efficient BS:

int k = 0;

for (int b = n/2; b >= 1; b /= 2) {

while (k+b < n && array[k+b] <= x) k += b;

}

if (array[k] == x) {

// x found at index k

}

* 1. lower\_bound: C++ method, returns a pointer to first array element whose value is atleast x.
  2. upper\_bound: returns a ptr to the first array element whose value is larger than x.
  3. equal\_range: returns both of those ptrs.
  4. Binary search can be used to find the lowest/highest value of a function as well.

1. Comparison: For 2 arrays

A = [5,2,8,9,4] and B = [3,2,9,5],

* 1. Going through each element in an array and then looking for it in another takes O(n2) time.
  2. Putting array A in a set and then going through each element of B and checking if it exists in set of A takes O(nlogn) time.
  3. The same as above but with unordered set takes O(n).
  4. Sorting both arrays and checking values at the same indices for both of them takes O(nlogn), nlogn for sorting and n for the comparison. This is the most efficient solution as sorting is done once but the rest of the alg is in linear time.

1. Complete Search: A general method to solve almost any algorithm problem, we look for all possible solutions using brute-force type approach and then select the best ones from them. It’s time taking and is a very general approach hence we may prefer a Greedy Algorithm or Dynamic Programming to solve a problem.
2. Data types:
   1. Binary Tree:

Height = log(n)

Best case time = B(n) = O(1)

Worst case : Searching for elements in leaf, so W(n) = O(logn) as logn is the height

Min. worst case time = logn

Max worst case time = n, as height is n when it is a skewed binary tree.

1. Factorial: The highest growing function, which is why it’s symbol is a ‘!’.   
   n! = n\*(n-1)\*(n-2)…(n-n+1)

A factorial series is 1, 2, 3, 6, 24, 120 … as we multiply all the previous values with the next position, which means it is a recursive function. It needs all the multiplications before the given number to calculate the next number.

Similarly, to go backwards, we divide with current position. So to go from 5! to 4!, we divide by 5. Which leads to 1/1 = 1 for 0!.

* 1. An interesting implementation of factorial is in simplifying multiplications,

For ex.

30\*28\*26\*24….\*2

We can see it’s a 30! but it’s having a gap of 2 numbers at each step.

So we can simply divide by 2, to get

15\*14\*13…\*1

which is just 15!. Now the divide by 2 for each step needs to be accounted for, so we multiply by 2. And as we know, we divided 15 terms,

30/2\*28/2\*26/2…\*2/2

which took 15 of 2’s or 215

And to balance the equation, we just multiply by 215

* 30\*28\*26\*24….\*2 = 15! \* 215

And to re-trace the steps,

x= 30\*28\*26\*24….\*2

x/2= 30/2\*28\*26…\*2

x/22= 30/2\* 28/2 \* 26…. \*2

…

x/215=30/2\*28/2\*26/2…\*2/2

x/215 = 15\*14\*13…\*1

x/215 = 15!

x= 15! \* 215

* Similarly, if we have odd numbers, we can use the even numbers’ solution to simplify them.

So, for 29\*27\*25…\*1

We can say it’s just 30! but it’s missing some terms, and we know the missing terms are 30\*28\*26….\*2 which we solved to be 15!\*215 so the solution is 30! / 15!\*215

Mathematically,

x = 29\*27\*25…\*1

x/30! = (29\*27\*25…\*1)/ (30 \* 29 \* 27 … \*1)

x/30! = 1 / (30 \* 28 \* 26 \* 24 … \*2)

x/30!/215 = 1 / 15!

(x \* 215)/30! = 1 / 15!

x\* 215 =30! / 15!

x= 30! / (15! \* 215)

* 1. Calculating Factorial in CP: It is highly inefficient to calculate a log in a programming language the normal way. For ex.:

100!/99!

is just 100 but for the language, it will first find factorial of 100 then divide it with 99 factorial and that is a humongous calculation.

This is why we use the gamma function,

<https://en.wikipedia.org/wiki/Gamma_function>

It’s implementation aside (which is very fast), it always gives us the factorial of the number-1 if the number is a non-zero positive integer.



In C++, we have the xgamma methods which are included in the base header itself but we can include math.h for additional overlods. There’s tgamma, lgamma, and then there generics. Basically lgamma returns loge of a gamma function, while tgamma returns the result of gamma function.

So for 5,

lgamma(5) would return 3.17, if we do exp(lgamma(5)) then it’d return 24.

tgamma(5) would return 24 (or 4!)

As it is visible, lgamma returns a very small value and using properties of exponentials we can simplify large factorial operations such as divisions into operations which wouldn’t result in an integer overflow. exp() returns the exponential (Euler’s number, 2.71828 raised to the the power of given number).

For ex.:

100!/99!

tgamma(101)/tgamma(100)

is one way of doing it and would just result in a humongous value. But,

exp(lgamma(101) – lgamma(100))

is a vehemently simple calculation in contrast to it. It is subtraction due to the property of exponentials which says ab/ac is just ab-c. Since the values theirselves are logs, the values are small.

1. Binomial Theorem:
   1. Polynomial: Many nomials, nomials are just terms in an expression.
   2. Binomial: 2 nomials, 2 terms.

For ex. x+1 is a binomial.

* 1. To break down binomials with powers,

Like

(1+x)0 = 1

Then,

(1+x)1= (1+x) \*1

(x+1)2 = x2+2x+12

pow3= pow2 \* pow1

= (x2+2x+12 ) \* (x+1)

= x3+2x2+x+x2+2x+1

= x3+3x2+3x+1

pow4 = pow3 \* pow

and so on

* They turn to polynomials.
* Another interesting point to note is that the sum of coefficients is

1 then 2 (1x+1) then 4 then 8 and so on, which is just 2k where x is the power of binomial. This can be helpful to check if the equation is right.

* Yet another interesting point to see are the coefficients theirsevles,

1

1 1

1 2 1

1 3 3 1

…

which is just the pascal’s triangle.

* In pascal’s triangle, the first row is the 0th row. Similarly, for each row the first column is the 0th column. To represent it symbollicaly, we use the nCr notation, where n is the row and r is the position in that row.
* Properties of Pascal’s Triangle:
  + nC0 = nCn = 1
  + nC1 = n
  + In a row in pascal’s triangle, the first floor(n/2) elements are unique but the rest are the same elements but in reverse order.

So,

nCr = nCn-r

This is aka the symmetry of a row.

* + 2n = nCr

as we know the sum of all elements in nth row is equal to 2n.

* 1. To open a binomial expression, we can use the pascal’s triangle.
* For ex, (x+y)5

Since the power is 5 we refer to the 5th row of Pascal’s triangle.

1 5 10 10 5 1

And then we simply add these after multiplying them with the terms,

1\*(x)5\*(y)0 + 5\*(x)4\*(y)1 + 10 \* (x)3\*(y)2 + 10\*(x)2\*(y)3 + 5\*(x)1\*(y)4+ 1\*(x)0\*(y)5

And that opens the binomial.

Generalization:

(a+b)n = nC0anb0 + nC1an-1b1 + nC2an-2b2 + …

nCnan-nbn

Here, nC0 means it’s the nth row of the pascals triangle and it’s the 0th coefficient in that row. There will be n terms after the formula where n starts at 0, so in integers it will be n+1 terms.

For an individual term,

Individual=**nCran-rbr**

Therefore,

(a+b)n = nCran-rbr

* 1. Factorial and Binomial Theorem: Binomials can be expressed as factorials as well. For ex.

6C0 = 1 , which can be written as 6! / 6!\*0!

6C1 = 6 , which can be written as 6! / 5!\*1!

6C2 = 15 , which can be written as 6! / 4!\*2!

Hence, a binomial can be expressed as factorial through this general formula:

**nCr = (n!)/ ((n-r)! \* r!)**

1. Permutation And Combination:

Probability= no. of favorable outcomes / total no. of outcomes

So for ex.

How many outcomes in a 4 digit pin?

10 possible digits \* 10 \* 10 \* 10

which is 104

And favorable outcome is a single PIN

so the prob. is 1/104

We multiply because they are related events, that is, we need first 10 digits AND we need the next 10 AND so on. But we add when they are separate events, like when we are concerned with what can be on the first digit, it can only be a single digit so we say the digit is x OR y OR something which just means, (1+1+1…) \* (1+1+1…)… so 10\*10… .

* + 1. Permutation: Ordered selection of possibilities. When the order of the outcomes matter we call the probabilities as permutations. This also means, permutations by default are non-repeating.

For the same 4 digit pin

The possibility of non-repeating digits

10\*9\*8\*7 as each step takes a sample space excluding the ones previously taken

which is another way of writing

10! / 6! as we don’t need 6 more digits

= 10!/(10-4)!

Generalizing,

n objects, which is 10 here

k positions, which is 4 here

nk would define the total outcomes for probability

And for Permutations,

n!/(n-k)! which is symbolically written as nPk

The formula is also called a npick, short for n picks for k positions.

When n==k, npick=n!

**npick= n!/(n-k)!** without repetition

**npick= nk** with repetition

* + 1. Combinations: Combination is a group of all the possibilities. Unlike Permutation, here order doesn’t matter. But what matters is that we don’t count repetitions and we don’t want identicals.

For ex. If we want to choose the no. of ways k objects can be put together out of n options, i.e. no repetitions and no replacements.

nPk / k! where k! is the overcounting.

This is because, when we say “order doesn’t matter”, we are saying treating all k object permutations as identicals, as we don’t care what each permutation is, we just want any combination of the k objects. So we divide by k! as k! is considered identical or with duplicates (like abc is not the same as cba in permutation but in combination it is considered a duplicate as all the elements exist in it).

npick / k! can be written as

(n!/(n-k)!)/k! which is of the form (x/y)/z, hence (x\*z)/y. (This is because, x^a is just x multiplied a times, say we were at x^2 and wanted to know x^1, we would just multiply with 1/x and x\*x/x becomes x or x^1. Similarly, x^0 is just x/x so 1, and when we get behind 0, it is just x^-1 or 1\*(x^0)/x, which is what defines x^-1 as 1/x. So when we have (x/y)/z, it can be written as x/(y/z) which is just x/(y\*z^-1) and after opening brackets, x/y\*z^-1 hence becoming x\*z/y. Alternatively, x/(y/z) can be multiplied with x/(y/z) \* (z/y)/(z/y) which becomes (x\*(z/y)) / ((y/z)\* (z/y)), denominator becomes 1 and numerator is left with x \* z /y.)

which is like **nCr = (n!)/ ((n-r)! \* r!)**

So it is just nCK

However, even though nCk is nPk divided by k!, it does not count for repetitions for objects. The k! is not eliminating repetitions, it is simply eliminating the permutations of the k objects.

When we say combinations, we say choose rather than arrangements for permutations. So nCk is just the no. of ways to choose k objects out of n options, but npick is the total no. of arrangements of k objects out of n options.

If we want to choose 3 objects from 10 options,

we say

10C3 = 10C7 (through the property of symmetry in pascals triangle, choosing 3 out of 10 and not picking 7 is the same as not choosing 7 out of 10 and picking the rest 3)

* Basically,

All Probabilities or Sample space is repeating and ignores identicals.

Permutations are non-repeating but ignores identicals. This non-repeating property ensures order.

Combinations are non-repeating and doesn’t count replacements.

A permutation can also choose to not count identicals, which becomes the same as a combination when there are only 2 different objects.

* + - * Identicals: When we take the permutation of a group of possibilities, we overcount. This is due to the fact that there are some orders that look exactly the same as others, since the normal formula doesn’t care about the value’s duplicates and counts all of the objects separately.

For ex.: KELLY,

How many possible arrangements of all the letters ?  
5! \* 4! \* 3! \* 2! \* 1!

which is 5P5

But as we can see, while the formula doesn’t care about the objects theirselves, there are 2 identical objects of ‘L’.

So KELLY != KELLY

and we say we overcount by an extra at each permutation.

Since it is 1 extra for each permutation, we can say the original is twice as big as it should be.

* 5P5/2 = 60

For KEELLY,

Similarly,

6P6/(2\*2)

But for KELLLY,

We now have 3 similar objects, and they can theirselves be arranged in 3P3 ways, which is 3!.

So,

6P6/3!

We can also say, that we are overcounting even non-repetitive objects, but since every single object can only be arranged in 1! ways, it doesn’t matter.

So,

6P6/1! \*1! \* 3! \* 1!

Generalizing,

For KELLY, 2 can be written as 2!.

When there are only 2 terms with equal counts, permutation becomes combination.

For ex.

KKKLLL

6! / (3!\*3!)

which is another way of writing

6!/ 3!\*(6-3)!

* 6C3

This is the case because binomials also have 2 terms,

(a+b)3

* a3+3a2b+3ab2+b3

This is just,

aaa + (aab or aba or baa) + (abb or baa or bab) + bbb

which is just all the permutations, if we replace a and b with K and L and increase power to 6, we can see all the possible permutations for KKKLLL, but this is only possible when K and L are equal in count. This proves, that for 2 terms with equal counts, permutation gives the combinations.

* However, when there are lesser positions than the elements to pick or choose from we can’t simply divide by the overcounting. This is because it is not certain that for each arrangement there is the repeated element.

For ex.:

ALGEBRA and pick/choose 3.

But when we do 7C3/2! and 7P3/2!, we get a fractional number and that does not make sense as there can either be a possibility or none, not in between.

So to solve it we divide it into sub-arrangements and then add all them up.

For Combination,

First, no As: 5 total letters and 3 positions so 5C3 = 10

Then, 1 A: This A takes a guaranteed slot so only 2 positions left, 5C2 = 10

Then, 2 A: Similarly to 1 A, 5C1 = 5

And total is 25, which is our answer. As we see different types of combinations, ones without A, ones with a single A and ones with one more A as that’s the total no. of As, and adding them up are the total combinations for 3 positions.

Alternatively, we could directly just take 1 A out and say ALGEBR, and in that case we just simply do 6C3, and this includes combinations with 1 A and no A. Which can be seen as it equals 20. Now we only need to see combinations with both As and when 2 A take up the slots, we are left with 1 position and 5 letters giving us 5C1 and total being 25.

Similarly, for Permutations,

First no As, 5P3 = 60  
Then 1 A, 5P2 = 20

However, since order matters here, the 1 A’s place matters as well, as we know

A \_ \_

\_ A \_

\_ \_ A

are the 3 positions, we multiply 5P2 with 3

Then 2 A, 5P1 = 5

Similarly, these 2 A’s non identical positons matter as well and that gives us 15.

Total is 135.   
  
Alternatively, we can directly take ALGEBR and that gives us 6P3 which is the sum of permutations achieved through 1 A and no As with order in mind, and it is indeed equal to 120 which we got from 5P3 + 5P2\*3. And we follow the same last step which gives us 15 and the total is 135.

* 1. (1+x)n + (1-x)n= 2 nC2k \* x2k for all +ve even values of n. – PC1
  2. Summing Permutations:

For ex.: Find sum of permutations of 1,2,3

Now we know there are 3! permutations, and in this case we can directly note them down and solve them but there is another way.

In default case we sum each column separately, then carry over and sum the next column and so on.

But alternatively, we can sum all the values of the objects individually and then sum them all up.

To do this,

first look at the objects, there’s 3 in this case, and when we create unique arrangements for all the objects, it is guaranteed that each column will have equal proportions of each object which can be obtained through n!/n and in this case that’s 2!. We divide by n because each object appears 1/n times in the whole value, if the object appeared k times in the whole value then we would divibe k by n, k/n. And we multiply it with all possibilities since it gets distributed equally in them.

Next we simply add the columns,

Sum of all of first object in 1s position = 2! \* 1 \* 1 (2! no. of 1s, value is 1 and position is of 1s)

Sum of all of first object in 10s position= 2! \* 1 \* 10

Sum of all of first object in 100s position = 2! \* 1 \* 100

Sum of all of first object = Sum of first object in all positions = 2!\*1\*(111)

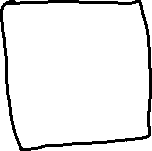
Similarly we do for the 2nd and 3rd object

Total = 2! \* (111) \* (1+2+3)

* 1. Paths across Grid:



For a grid:



1. Total paths from A to B
2. Total Paths from A to C then B
3. Total paths if row of A is non traversable
4. Total paths if row of X is non traversable
5. If we observe a path, it is RRRR DDDDDD

and then if we observe another path

DDDDDD RRRR

or

DDRRDD RRDD

we can see that there are only limited rights and downs and the difference in each path is of order, so we know it is a question for permutations.

Now, we can see there are 10 slots from a path which means there are 10! ways to go from A to B, but we can also see that we have overcounted,

RRRR and DDDDDD

The Rs and Ds even if swapped will create identicals which we don’t care about. So we have overcounted by row! \* column!

so total ways is 10! / (4! \* 6!)

1. We can solve this by multiplying 2 paths, A to C then C to B.

A to C:

4! / (2!\*2!)

C to B:

6!/ (2! \* 4!)

Total ways = A to C \* C to B

We multiply because each path taken from A to C will occur C to B times in the overall permutation. In other words, if A to C has 1 path, then adding this path to C to B means this 1 path has to be taken only once from A to B. But multiplying this 1 with all paths of C to B means that this 1 path occurs for each path from A to B, as we know, there are multiple paths from C to B and for each one of them, a path from A to C is needed otherwise A to B can’t be fulfilled, so multiplication is indeed needed. Similarly if there were 2 then it’d mean that for each path from C to B, there are 2 ways for A to reach C. Multiplying means dependency on the previous permutation while adding means irrelevance.

1. If row of A is non traversable then we only have 1 path and that is down to X, and from X to B there are 9!/(4!\*5!) ways. So the total is 1\*(X to B)
2. If the row of X is non traversable then we can just use the row below. So A to P is 1 way, then from P to B we have 8!/(4!\*4!) ways. Next, from A to Q we have 2 ways, and Q to B we have 7!/(3!\*4!) ways and so on

Total = 1 \* P to B + 2\* Q to B + 3 \* C to B + 4\* R to B + 5\* S to B

* 1. More spots than available objects:

In this case, repetition is guaranteed. Hence we can’t do permutation and instead we solve it like normal probability but with combination.

For ex.:

3 letters, A,B and C and 4 spots

So the total ways are 3\*3\*3\*3 or 81 for all probabilities.

1. With 2 of Bs,

2 B = 4C2 \* 1

as we need to choose all probabilities with 2 slots out of 4 and there is only 1 object to put in them.

The remaining spots = 4-2 = 2

And the remaining choices, 2, A and C

So the total ways for remaining spots = 22 where 2 of base are the available choices and the power is the number of spots.

And the total ways for all the spots = 4C2 \* 1 \* 22

Generalizing,

22 can be written as 2n-2 where n are the total no. of spots.

and the form can be written as,

Total ways = nCk \* o \* mn-k

where n are the total no. of spots, k are the spots needed, o are the objects to put in them and m are the remaining objects.

1. n letter word consisting of A,B and C with 0 to even no. of Bs = 1/2(3n+1)

spots= n

0 B = nC0 \* 1 \* 2n

2 B = nC2 \* 1 \* 2n-2

4 B = nC4 \* 1 \* 2n-4

.

.

All Bs= nCn \* 1 \* 2n-n = 1

Finally, we add all possibilities

0B + 2B + 4B + …. +nB

nC0 \* 2n + nC2 \* 2n-2 + nC4 \* 2n-4 + … + nCn \* 20

To get this in the form of ½+(3n+1) we need to use the formula PC1.

This can be made into the Sigma part of PC1, but we need to simplify it first.

Since we know, nCr = nCn-r,

nC0=nCn

nC2=nCn-2

and so on

nCn \* 2n + nCn-2 \* 2n-2 + … + nC0 \* 20

which can be turned into a simple sigma

nCk 2k

K goes till n/2 because K doubles at each step so it will reach n in n/2 steps.

x=2 in this case and this is of the Sigma form of PC1 without the coefficient 2, which means the first part becomes ½ \*[ (1+x)n + (1-x)n]

Since the equation is equal to Sigma form of PC1, it must also be equal to the other form.

* ½ \*[ (1+2)n + (1-2)n]

½ \*[ 3n + (-1)n]

since we know n is all +ve even integers, -1 will turn positive.

½ \*[ 3n + 1]

Hence proved.

1. a
   1. More spots than available objects but no repetition:

For ex.:

2 eggs and 6 spots to put them in. But each egg can only be placed once.

In this case, we flip the operation.

So, we instead look at 6 objects and 2 spots, because we can also say that the there can be multiple spots for a single egg.

So like spot 1 for egg 1 and spot 5 for egg 2 and so on.

So,

\_ \_

6 5

and that is 6\*5 or in other words, 6!/(6-2)!

so 30 permutations.

Since they are just 2 eggs they are identical. We remove identicals the same way,

30/2! = 15

So, 15 combinations.

1. Bitwise Operations: Can be used for some interesting properties.
   1. XOR: XOR can find missing and repeating numbers. For ex.:

In an array 1,2,3,5,0

we know 4 is the missing number, to get it we use the xor’s property of cancelling out same numbers (2^2=0)

So,

1^2^3^5^0 ^ 1^2^3^4^5

leaves us with 4 as the rest cancel each other out and 4^0 is just 4.

and for repeating number

1,2,3,4,4

1^2^3^4^4 ^ 1 ^ 2 ^ 3 ^ 4 ^ 5

leaves us with 4^5 = 3

Now, we know that XOR only gives 1 when exactly 1 bit is set in an operation. So the rightmost 1 bit in n from a^b =n is the bit that differentiates a from b.

Now that we know what differentiates a from b, we just find all the viable candidates for a and then b, namely, all the values in the array where the given bit is 1 goes into a box and all the values where the bit is 0 go into another. And any 1 of the values in the box with the given bit 1 is the missing value, this is because the bit that turned 1 exists because it was added and wasn’t cancelled out, and indeed, we added the 1^2^3^4^5 which brought 5 and while all the other values got cancelled, it remained.

Because the bit with 1 is the new value, we can conclude that the other value must be the repeating value, this is because there can only be 1 or 0 and we know what occupies the 1 as we have defined it, meaning the only value for the 0 is the one that is not 1, i.e., not the missing value.

Hence we get,

1 ^ 3 ^ 1 ^ 3 ^ 5 = 5 2^4^ 4^ 2 ^ 4= 4

and we have our repeating value with the missing value also separated out.