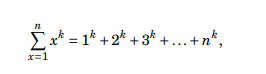
**Competitive Programming**

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. 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. 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

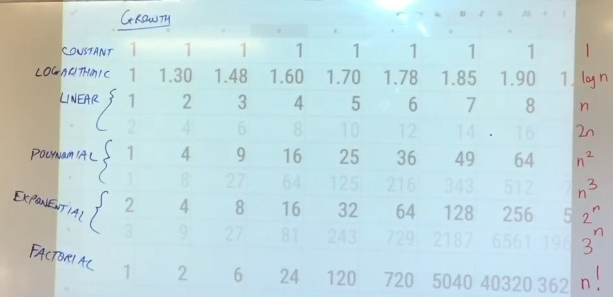
Since p++ is ran for log2n times Total TC = log2log2n

* 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.

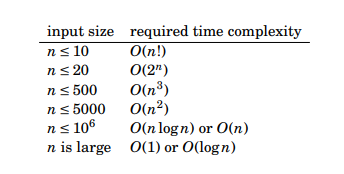
nk: k loops

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. 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. Bits: Can be used for some interesting properties.

Left and right are flipped in bits.

The rightmost bit is aka the Most Significant Bit as changing it gives the biggest change to the number. More specifically, if we turn a bit at position k to 1 from 0, the value of the number is added by 2k-1 and if we turn it to 0 from 1 then it is subtracted by the same.

The leftmost bit is known as the Least Significant Bit as changing it causes the least change in the value. Similarly, the start and last bits are flipped too, the start bit is the LSB and the last bit is the MSB.

For ex.:

0 1 1 0 = 6

1 1 1 0 = 6 + 23 = 14

* 1. Value:

Each bit can have 2 integer values, meaning 2 bits have 22 values and so on.

For ex.:

\_ \_

are 2 bits and they can have 4 values. 3 positions give 8 values and so on.

* 1. Binary addition and Subtraction:

Addition:

1 + 1 = 10

0 + 1 = 1

0 + 0 = 0

1 + 0 = 1

Subtraction

1 – 1 = 0

0 – 1 = 1 (1 borrowed from next element, 0 – 1 basically becomes 2-1)

1 – 0 = 1

0 – 0 = 0

if carry over crosses another 0 then it makes it 1,

For ex.:

100

- 011

0 0 1

The first 1 is a result of 2-1, the next 0 is a result of 1 – 1 as 1 had been carried over it from the next element to the previous element, lastly 0 – 0 which gives 0.

If there isn’t a bit to borrow 1 from, then it carries over 1 from infinity.

0 0 0

0 0 1

1 1 1

which is 7, and we discard the additional bits as 3 bits were what we wanted.

As we can see, we have wrapped around. Hence we can say that 0-1 is equivalent to 2n-1, we can prove that as well.

If we subtract 1 from two’s complement, it will still give the same answer,

0 0 1

+ 1 1 1

* 0 0 1

= 1 1 1

So, to the computer, subtracting a number from 0 is the same as flipping all the bits, adding 1 and then subtracting the number. Which is what makes 2’s complement the same thing.

* 1. Binary multiplication and division:

0 × 0 = 0

0 × 1 = 0

1 × 0 = 0

1 × 1 = 1

And then just like decimal multiplication, we follow the same steps.

1 0 1 1

x 1 1

1 0 1 1

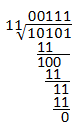
+ 1 0 1 1 0

1 0 0 0 0 1

we put a 0 in the second line to indicate the 10s place.

Division:

Generally a bit more complex to grasp but it’s just like normal division.



* 1. Signed and unsigned bits: In C++, an integer is represented by 32 bits on 32 bit platforms.

For ex.:

11111111111111111111111111010101 = -43

This is a signed int in binary, where the MSP is a bit that denotes if the value is negative(1) or positive(0). So the actual range of values is -231 to 231 – 1 where -1 is done to exclude 0. A signed int can hold negative values, but to do so the bits are in 2’s complement (flip the bits and add 1). The 2’s complement is a shortcut to allow quicker operations, and unifies addition and subtraction.

If we take even the sign bit then it increases the range of values to 232-1 but since we took the sign bit, it will only give values from 0 to the limit.

Following the property of subtraction of binary values, a signed int -x equals 2n-x in unsigned int.

For ex.:

int x = -43;

unsigned int y = x;

cout << x << "\n"; // -43

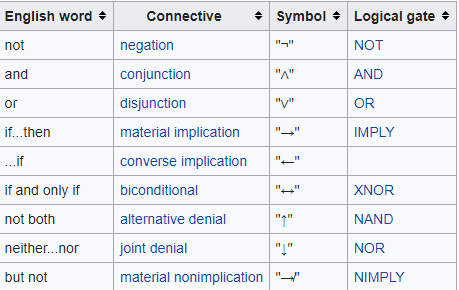
cout << y << "\n"; // 4294967253

In signed values, any value after 2n-1-1 is -2n-1 and for unsigned values it is 0.

This is because of binary value wrap around as shown earlier.

* 1. Logical Connective: A logical connective is a constant that connects 2 logical arguments, or in other words, it joins 2 parts into 1 with a relationship. In binary, the connectives are known as logical gates as they can block/allow bits.

Here’s a mapping of connectives to logical gates:



There’s also XOR gate.

Since binary only has 2 values, if something is A then it’s not B or if it’s not B then it’s A and so on.

Not: Flips the bit, as not A is B. Requires exactly 1 input. Represented as

S = A or S= ~A

ANSI Symbol:



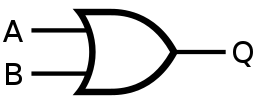
And: 1 if both A and B are 1. 0 otherwise.

 or S = A B



Or: 1 if either A, or B or both are 1. 0 otherwise.

S= A ∨ B



Imply: If A is 1 follows the value of B otherwise 1. Basically, always 1 unless A is 1 and B is 0.

S = A B



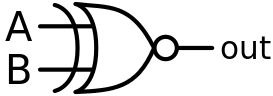
XOR: 1 only when A or B is 1, not both, not neither.

S = A ⊕ B



XNOR: 1 if both A and B are same value. aka Equivalence Gate.





NAND: The opposite of A AND B. So, it is 0 if both A and B are 1, 1 otherwise.

S= 



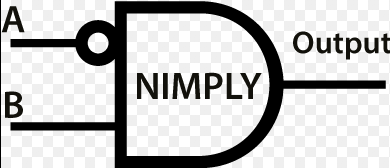
NOR: The opposite of A OR B. So, 1 if both A and B are 0, 0 otherwise.

S = (A ∨ B)



Nimply: 1 if A is 1, 0 otherwise.





* + 1. Buffer: A simple gate that stores the input and sends it back.

1 if A is 1 0 otherwise. Single input.

S = A



* + 1. Every gate can be represented with NAND or NOR gate. That’s why they are called universal logic gates.
    2. Precedence: If an operation involves multiple operations, a precedence is followed, which is



Overridden by brackets just like VBODMAS.

So, A Or B And Not R S is interpreted as (A Or (B And (Not R))) S

Precedence is required because logic gates can only work for 2 inputs excluding some that can only work for single input.

* + 1. Bitwise operators: The gates are the fundamental building blocks of transistors in the CPU as all binary operations are performed using them. Most programming languages directly understand logic gates. They are known as so.   
       In C++, they are

And: &

Or: |

Xor: ^

Not: ~ . In C++, Not A = - A – 1, ~29 = -30.

<<: Left Shift

>>: Right Shift

* + 1. Set vs Bit operations:

a Intersection b : a & b

a union b : a | b

complement a : ~a

a difference b : a & (~b)

* + 1. Bit shift operators: In C++ Bit shift operators shift all bits to the left or to the right by x times.

For ex.:

6 >> 2 = 1

right shifts bits of 6 2 times.

So,

0 1 1 0 >> 2

is

0 0 1 0

then

0 0 0 1

Right shifting discards the bits overflowing on the right (LSBs) and adds the new MSBs as 0.

Similarly,

6 << 2 = 8

left shifts bits of 6 2 times.

0 1 1 0

is

1 1 0 0

then

1 0 0 0

which is 8.

But left shifting is a bit different from right shifting, as unlike right shift, left shift is affected by the size of the binary container, for 4 bits, 6<<2 is 8 but for 5 bits it is 24 and so on.

* 1. Bitmask: A bitmask is used to access single bits of numbers. The logic is pretty straight forward, we & the number with 1 and get 1 if the bit is 1, and if we left shift 1 and then & it with x then we can essentially get any bit given the 1 is left shifted to a position that’s valid in the x.

That is, a bitmask for accessing single bits of numbers is x & (1<<k) where x is the number whose bit we want and k determines which position bit we want, which would be the k+1 bit from the left (right and left are flipped in bits).

For ex.:

6 & (1<<2)

is

0 1 1 0 & 0 1 0 0

will return

0 1 0 0

which is 4 in decimal

as the k+1 bit from left is 1.

There are other bitmask operations too,

x | (1<<k) returns x but the k+1 bit is set to 1.

x & ~(1<<k) returns x but the k+1 bit is set to 0.

x ^ (1<<k) returns x but the k+1 bit is flipped.

x & (x-1) returns x but the last bit is set to 0. Last is the MSB, so this flips the MSB.

For ex.:

6 & (6-1)

6 & 5

0 1 1 0

0 1 0 1

1 1 0 0 = 12

x & -x sets all the 1 bits to 0 except the last 1 bit.

x | (x-1) inverts all the bits except the last 1 bit.

x & (x-1) = 0 when x is a power of two.

* + 1. int: By default, all bit operations are performed on ints as in C++ raw numbers are inferred as ints. When x is not an int, it is advised to not use int bitmasks, like if x is long we do x & (1LL<< kLL)
    2. Additional bit methods provided by G++ for C++:

• \_\_builtin\_clz(x): the number of zeros at the beginning of the bit representation

• \_\_builtin\_ctz(x): the number of zeros at the end of the bit representation

• \_\_builtin\_popcount(x): the number of ones in the bit representation

• \_\_builtin\_parity(x): The parity (even or odd) of the number of bits with 1 in the bit representation.

These methods accept ints and the LL version is suffixed with the same.

* 1. Binary to Decimal:

For ex.: 1 0 0 0

in decimal is 8, as

23\*1 + 22\*0 + 21\*0 + 20\*0 = 8

This is because each bit holds 2 values, and the importance of the value is incremented with the position, so the first bit can only represent 2 decimal values, 0 and 1, the next can represent 4 values, 0, 1, 2 and 3 and so on as the bits can represent twice more values with each extra bit.

Decimal to binary:

9 to binary

The opposite way, here instead of multiplying we divide by 2. As the more a number can be divided by 2, the bigger it is. So

9 = 23 + 1

or we can divide and hold the remainders,

9/2 = 1

4/2= 0

2/2 = 0

1/2 = 1

So, 1001 for 9, going from bottom to top.

Similarly,

8 = 1000

8/2 = 0

4/2 = 0

2/2 =0

1/2 =1

* 1. Even/Odd to binary: One interesting property of binary is that since it holds 2 consecutive values per bit, one of them will be even and the other will be odd. This is because in integers, every 2nd number is even, because it has appeared at 2k position, like 22 steps after 0 so it is bound to be divisible by 2, as 2 steps have passed.

So, in binary, the steps of 2 fit into the first place or the LSB and hence by checking the LSB of a binary number, we can determine if it represents an even or an odd decimal number.

As it starts from 0, like 0000, and 2 steps from it is 0010 we can conclude that 0 at LSB means even numbers.

Thus, to easily check binary we can simply do x & 1 (to check if LSB is 1), and if its 0 then it is even otherwise odd.

Generalising, we can say that x is divisible by 2k exactly when x & (2k-1) = 0

* 1. Sets and bits: A set is a collection of unique numbers or objects. Using just 1 binary number of size n where n is the largest number of the set, we can represent all the values of the set. We do this by simply setting the bit at position of the number as 1 in the binary.

For ex.:

S= {1,3,7}

binary number is,

b= 0 0 0 0 0 0 0

and then we put 1 on the positions where the position == value

b = 1 0 0 0 1 0 1

Then to simply get all the values of the set, we only need to iterate over the binary and keep track of index

for i=0, i<7; ++i

if(b[i] == 1)

print(i+1)

and we get 1 3 7 back.

To do the same in C++,

int x {0};

x | = (1<<1);

x | = (1<<3);

x | = (1<<7);

for(int i {0}; i< 32 ; ++i)

{

if(x & (1<<i)) cout<<i<<” “;

}

prints 1 3 7

Do keep in mind that in c++ the int is 32 bit so use bitset or vector<bool> or long long for bigger sets.

* + 1. Traversing through subsets: Just like the loop above traverses the elements of the set, there are other methods to see the subsets.

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

…

}

Same as i < 32 when c++’s ints are concerned. But n is from 2n, so if we want i < 32 we would go i < (1<<5).

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

if(\_\_builtin\_popcount(i)== k)

{ … }

}

Goes through subsets with exactly k elements.

So if n =3, k =2, it’d be true for 3 ,5 and 6.

int b{0};

do {

…

} while(b= (b-x)&x);

Goes through subsets of x. Assignment is an expression in C++ so it returns true for all values except b=0.

* 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.

* 1. a

1. Basic Program:

#include <bits/stdc++.h>

using namespace std;

int main() {

// solution comes here

}

and ran with

g++ -std=c++11 -O2 -Wall test.cpp -o test

std defines the compiler to be used, -O defines the optimization level, -W defines warning level, test.cpp is our file and -o <file> is the output file name.

bits/stdc++ in g++ includes all standard library headers into the program.

* 1. We can use the following methods to improve cin/cout time.

ios\_base::sync\_with\_stdio(0);

cin.tie(0);

The first method decouples c and c++ i/o buffers. The second method decouples cin and cout and that means

We may also use \n instead of endl. endl flushes the stream which takes time but \n doesn’t.

* 1. For file i/o, we can use

freopen("input.txt", "r", stdin);

freopen("output.txt", "w", stdout);

for quick r/w.

* 1. Data Types ranges:

Int: -231 to 231 or -2\*109 to -2\*109

long long: -263 to 263 or -9 \* 1018 to 9 \* 1018

Double

long double

* 1. Mod: The numbers can be very large, to counter this we generally use x % m where m is 109 + 7. Mods have some interesting properties, which allows our numbers to be minified.

(a + b) mod m = (a mod m + b mod m) mod m

(a − b) mod m = (a mod m − b mod m) mod m

(a · b) mod m = (a mod m · b mod m) mod m

and the basic division rule,

17 % 5 = 2

17= (3\*5) +2

or

x % m = n

x= ((m-n) \* m ) +n

So for ex.:

To find 24 – 15, if 24-15 is a too large number, we can get its value to fit in an int using mod, in this case lets just take mod as 3

(24 – 15) % 3 = (24 % 3 – 15% 3 ) %3

* (0 – 0) % 3 = 0
* 9 % 3 = 0

and we simply store 0 and 3. Then to get the original value back, i.e. 9 in this case.

x = (3\*3) + 0

x = 9

and we have x back.

A subtle issue with mods in C++ is that -ve numbers when modded either give 0 or a -ve value. To fix this we simply add m to the remainder,

x=x%m

if (x<0) x+=m

* 1. Comparing floats: Comparing floats is an error prone task as rounding errors are too common.

To counter them, we take a small margin and if the difference is within that margin, we can say the values are equal. The margin is generally defined as ε, where ε is a small number 10-9

For ex.:

if(abs(a-b) < 1e-9) {

//a and b are equal.

}

1. Recursion: Recursion, programmatically, is simply when a function calls itself. But the overall idea behind recursion is to divide and conquer. It propagates the input and keeps propagating the input after an operation until a base case is reached (if there isn’t any or is never reached then we have an infinite recursion).

For ex.:

We can have a singular branch recursive tree, in it, only 1 branch is propagated.

int add(int num)

{

if(num==10)

return num+1;

else

add(num+1);

}

add(1);

This is basically how single branch recursions look and there’s not much to do with them as they are just like loops. But unlike loops, recursion tree can only be a set levels deep (depth of function call stack). As unlike loop, each time the function calls itself the program goes into 1 level deeper in the call stack and the function awaits the return value. In C++ the max. call stack size, which defines the max recursive depth, is dependent on the OS which still gives us a very large depth.

Multi-branch recursive tree

int add(int num)

{

if(num == 3)

return num+1;

else {

auto val1{add(num+1)};

auto val2{add(num+1)};

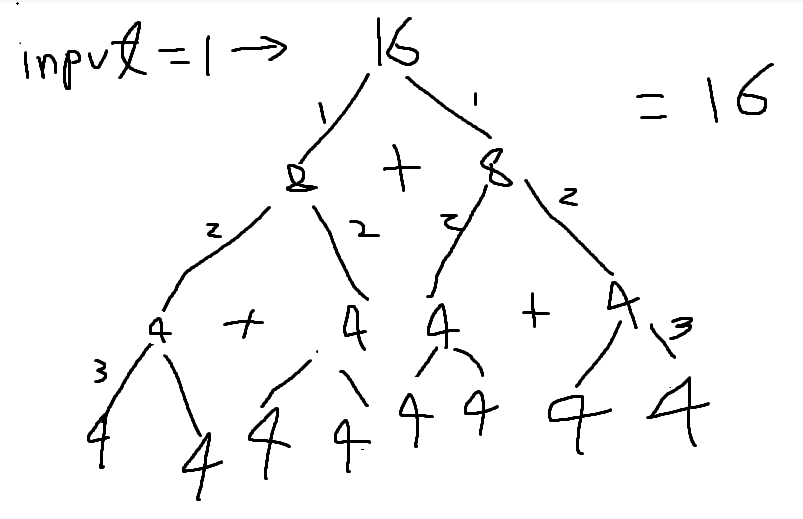
return val1+val2;

}

}

add(1);

returns 16. As this one branches and goes into multiple branches.



This is how it looks, read it from the bottom, as that is the base case, called with 3 and returning 3+1. Then it adds the values up and goes back, finally popping 16 out of the function.

Similarly, if we input 16 but reverse the signs and subtract instead of add, we will reach base cases where we have 4, this has subdivided the big number into many smaller numbers. This subdivision property is very useful in divide and conquer.

1. C++ G++ configs:

To compile a cpp file,

g++ -std=c++11 -O2 -Wall test.cpp -o test

compiles using C++11 standard and generates an executable ‘test’.

Assembly code:

g++ -S test.cpp -o test.out

Outputs the machine code.

* 1. Compiler Optimizations: Optimizations are done on release builds.

A lot of things like manual bit operations and bit to check even/odd etc. are faster than normal operations. We can manually use them in the code but we don’t need to as compilers do a variety of optimizations which turn things faster.

Another optimization is removal of unused code, so if a method is not used anywhere, it will not be included in the binary.

* 1. Hardware Specific Optimizations: G++ flag -march=native enables hardware specific optimizations. Such as popcount, which is a method that returns the number of 1s in binary representation of a number, is available through a special instruction popcnt to many processors and hence if we use it, our program can become a lot faster.

These optimizations are turned off by default as not all hardware support the same optimizations and when the generated binary is ran, there is no guarantee it will work on all processors.

Another alternative to enabling hardware specific optimizations, is by defining the arch in the cpp file itself.

#pragma GCC target ("arch=sandybridge")

For G++, this directive declares that the target arch is sandybridge and hence it can apply all hardware level optimizations supported by the arch.

* 1. Caches: It is important to keep the general CPU architecture in mind when designing algorithms. This is because things like the cache greatly affect how fast a piece of code is.

Caches in a CPU are divided in multiple layers, with each layer growing slower but bigger. Generally we have 3 layers, L1 cache, L2 cache and L3 cache with L1 being the fastest and smallest.

When CPU fetches data, it fetches a whole block of memory to the cache, so basically when we access a single element of a vector, we get adjacent parts or the whole vector in the cache for free and any work on them will take almost no additional time.

So

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

s += x[i][j];

}

}

is a whole lot faster than

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

s += x[j][i];

}

}

* 1. Parallelism: Code that doesn’t depend on each other, can be executed parallely by the CPU. This happens automatically. So,

ll f = 1;

for (int i = 1; i <= n; i++) {

f = (f\*i)%M;

}

and

ll f1 = 1;

ll f2 = 1;

for (int i = 1; i <= n; i += 2) {

f1 = (f1\*i)%M;

f2 = (f2\*(i+1))%M;

}

ll f = f1\*f2%M;

are the same thing but the 2nd code is executed a lot faster as the CPU can work on 2 independent variables at the same time for each loop.

1. Sorting:
   1. Sorting Algs: It is advisable to use the predefined sorting algs where-ever possible but we can create our own as nevertheless.
      1. Bubble Sort: Takes O(n2) time. Works by comparing each consecutive element and swapping them if they are inverted (For <, a>b when arr[i]<arr[j])

This runs for n times for n elements.

for (int i = 0; i < n; i++) {

for (int j = 0; j < n-1; j++) {

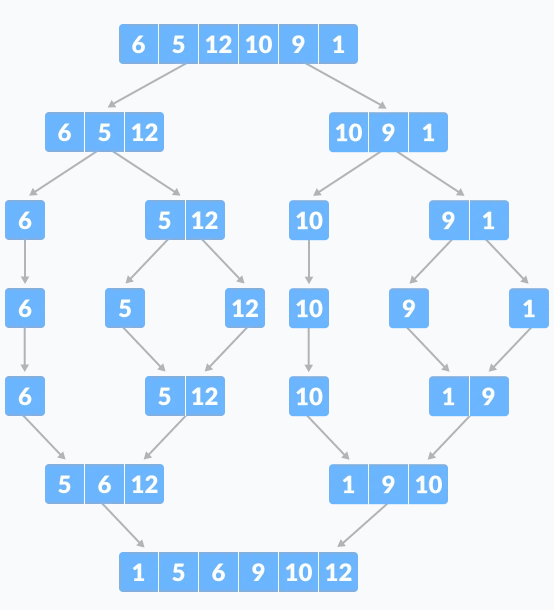
if (array[j] > array[j+1]) {

swap(array[j],array[j+1]);

}

}

* + 1. Merge Sort: Works using the divide and conquer approach. Here we subdivide the array and sort just the small parts theirselves. It takes O(nlogn) time. This is because at each step the input size halves, so the TC is 2k=n or logn and the merge step takes TC n so the total is nlogn.



The algorithm for merge sort is a bit complex as it involves recursion.

<https://github.com/cryoelite/CompCode/blob/main/DailyProg/merge_sort.h>

* + 1. Counting Sort: All sorting algs that rely on comparing elements of the array have a TC omega(nlogn). But if they don’t compare their elements and are rather affected by an external variable that just iterates over the elements then we can have a lower TC such as in Counting Sort. In counting sort we keep a bookkeeping array, which basically counts the occurrence of a given few elements in the array and then generates a new array out of the elements. It has a TC of O(n).
  1. Using sorting, a lot of algs can reduce their TC. This is why we generally sort the array input before working on it.
     1. Sweep Line algs: These algs model a problem as a sorted set of events. Then they ‘sweep’ through the sorted array and reach a solution.

For ex.:

n customers’ arrival and leaving time is given in an array, find the max. no. of customers at any given time.

So, say the customers are A B C and D and their times are [3,6], [4,5], [7,8], [1, 5]

Then a solution here is to simply sort by the first element,

[1,5], [3,6], [4,5], [7,8]

Then we

Then maintain a counter and a maxCounter, if a customer arrives, increase it and if they leave decrease it. Keep track of the max of counter values.

counter=0;

tempTime=0;

maxCounter=0;

for elem in arr:

if(elem[0] == tempTime || elem[0] == tempTime+1) {

++counter;

}

if(elem[1] == tempTime || elem[1] == tempTime+1) {

--counter;

}

maxCounter= max(counter,maxCounter);

tempTime+=2;

It sweeps through the intervals.

* + 1. Greedy Algs: These algs sort the input and then greedily construct the solution, i.e., they try to find a solution rather than trying to get the best solution.
  1. Binary Search: 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. Binary search is simply a searching method in which we discard half of the input recursively until we find find our element, i.e., we take the start and end, check if it’s the element and if not then if it’s greater or lesser, if lesser we make the mid the new right end and go to the mid between these new points, if greater we do it with start.

int k = 0;

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

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

}

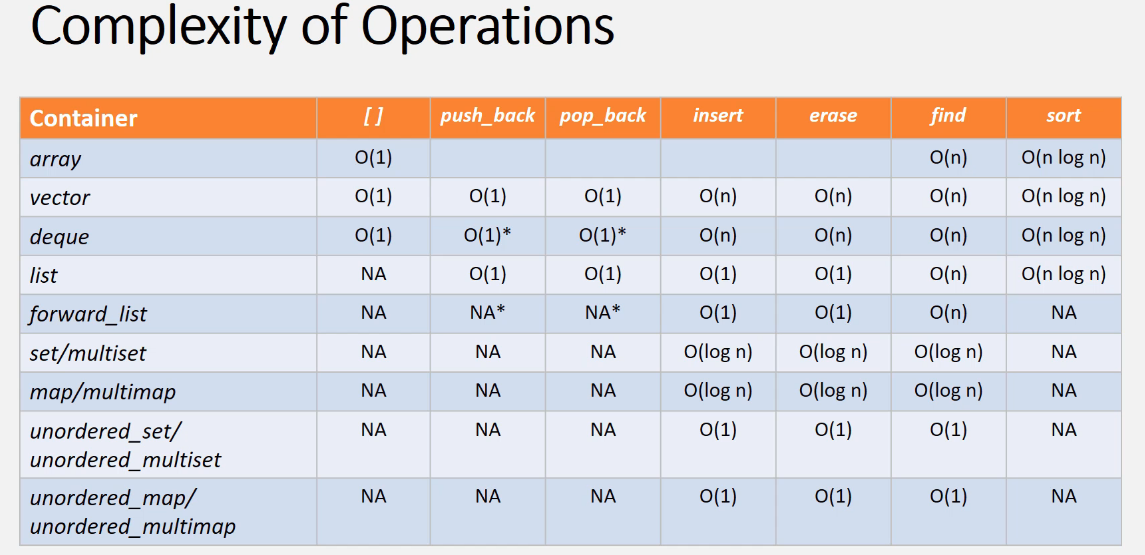
if (array[k] == x) {

// x found at index k

}

1. Data Structures:

Basic TC: For C++ inbuilt 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. Iterators and Ranges: Iterators are pointers to elements in the vectors, sets etc.

C++ uses iterators for all operations on vectors and as such the algs that accept vectors or range of values, accept iterators.

* 1. Dequeue: Just like vector but also allows push/pop from the front. It’s short for doubly ended queue. While deque scales the time complexity same as vectors they have larger constant factors, which is why they mustn’t be used without need.
  2. Stack<T> and Queue<T>: Based on dequeue. Stack LIFO has push, pop and top methods. Queue FIFO, push, pop, front and back methods. Both are available in GCC but not MSVC.
  3. Set: Optimized for insertion (<set>.insert(<T>)), search(<set>.find(<T>), if result != <set>.end() elem found) and removal (<set>.erase(<T>)). set is based on balanced binary search tree and the operations take O(logn). C++ also provides unordered\_set which is based on hash table and operations in it take O(1) time.

Sets are used for their important property of only having only unique elements, if an element already exists in a set, it is not inserted. Hence, <set>.count(<T>) always returns 0 or 1. Set is ordered, meaning the order of elements is preserved.

unordered\_set doesn’t preserve the order so each time we traverse the unordered\_set, the order could be different.

* 1. Multisets: Just like set but allows multiple values. There’s unordered\_multiset too. Everything is the same except count and erase take longer to work by a factor of O(k) where k is no. of copies of the element.
  2. Map: K-V pair array. Just like Set, Map is based on Balanced Binary Search Tree. And just like unordered\_set, unordered\_map is based on hash table. The TC is same as Set too.
  3. Priority Queue: It is a multiset based on a heap structure, which is a special type of Binary Tree. So just like a multiset elements are sorted on storage. However, insertion and removal take O(logn) and retrieval takes O(1) in a PQ. Unlike the traditional multiset, PQ has much smaller constant factors so it is better to use PQ if we only need to find min and max elements.

PQ has no iterators and the only way to access elements is in a queue approach (with top()) on top of sorted elements.

For ex.:

#include<queue>  
  
priority\_queue<int> pq{};

pq.push(6);

pq.push(2);

pq.push(5);

pq.top();// returns 6

pq.pop();

pq.top(); //returns 5

. . .

priority\_queue<int, vector<int>, greater<int>> pq2; //sorted in ascending order

* 1. Policy-Based Sets: They are a part of GCC. To use them we need to include,

#include <ext/pb\_ds/assoc\_container.hpp>

using namespace \_\_gnu\_pbds;

They have some special types like indexed\_set. Indexed Set (non generic, only for ints) is a typedef on an integer tree<…>. This set also has indexing for sets so <indexed\_set>.find\_by\_order(2); returns an iterator to the element on index 2. Similarly, <…>.order\_of\_key(2); returns the position of the given element, if the element doesn’t exist in the set, it returns the position where the element would be if inserted. Both of these methods have O(logn).

1. Practical differences between Data Structures: Time complexity simply measures growth not the actual time taken. While it is generally more than enough to determine good algs, it is always a better idea to check actual time taken as well.
   1. Set vs Sorting: Sorting is ~10x faster than unordered\_set which is ~2x faster than normal set.
   2. Map vs Array: Array/Vector is ~100x faster than unordered\_map which is ~3x faster than normal map.
   3. PQ vs multiset: If we don’t need to access all elements and only the min/max then PQ is a much better data structure than multiset. It is ~5x faster for working with elements on the tips.
2. Dynamic Programming: A method to efficiently find solution to a problem by breaking it down into subproblems and finding solution to the subproblems. By recursively solving all the subproblems we can get a solution to the main problem in DP.
   1. For ex.: The coin problem aka change-making problem. We are given a set of coins and a target sum, and we have to find the minimum no. of coins needed to construct a sum for it. The coins can be repeated.

We can construct a greedy solution to it by taking the largest value coin <= target value coin and then adding the coin until target is reached or target is < coin + same coin then we simply find the largest value coin <= target again and continue. But this solution isn’t optimal, as for example if set is {1,3,4} and target is 6 the optimal solution is 3,3 or 2 coins but greedy would result in 4,1,1 or 3 coins.

To solve it with DP,

first we understand the function normally,

we need a function solve(x) that returns the min coins for an input {1,3,4}

solve(0)=0

solve(1)= 1

solve(2)=2

solve(3)= 1

solve(4)=1

solve(5)=2

…

From the function we can observe that for a value x, there can be max 3 coins making it up, and we can solve a smaller value then calculate a bigger value from it, so

solve(x)= min(solve(x-1)+1, solve(x-3)+1, solve(x-4)+1)

This equation says, solve(x) has a solution if we solve for the first coin and then recursively solve it, or the 2nd or 3rd one and the smallest among the 3 is the optimal solution. The base case here is solve(0) as 0 needs 0 coins. We add +1 to include the base case in the value.

So for the coin problem the values are defined as

solve(x) = INF for x<0

0 for x=0

min of c in solve(x-c)+1 x>0

where c is a value of a coin.

x<0 is infinite because no sum of positive values can reach <0.

Now we simply represent the equation in C++,

auto coins{vector<int>({1,3,4})};

int solve(int x) {

if(x<0) return INF;

else if(x=0) return 0

else {

auto best{numeric\_limits<int>::max()};

for(auto &c: coins)

{

best = min(best, solve(x-c)+1);

}

return best;

}

}

This finds an optimal solution, however it will run recursion on a lot of values that have already been calculated which is a waste of resources, so we use a technique called memorization. Memoization refers to ‘remembering data’, so if the function can remember calculated values (efficiently) then a lot of resources can be saved.

We do so here with,

auto ready{vector<bool>(N)};

auto values{vector<int>(N)};

where N is a constant big enough to create a vector where all values can fit.

then we simply use it,

auto coins{vector<int>({1,3,4})};

auto ready{vector<bool>(N)};

auto values{vector<int>(N)};

int solve(int x) {

if(x<0) return INF;

else if(x=0) return 0

else if (ready.at(x)) values.at(x);

else {

auto best{numeric\_limits<int>::max()};

for(auto &c: coins)

{

best = min(best, solve(x-c)+1);

}

values.at(x)=best;

ready.at(x)= true;

return best;

}

}

Wherever recursion is concerned, iteration can be used as well. To iteratively solve the coin problem,

value[0] = 0;

for (int x = 1; x <= n; x++) {

value[x] = INF;

for (auto c : coins) {

if (x-c >= 0) {

value[x] = min(value[x], value[x-c]+1);

}

}

}

Iterator implementation is faster too as it has no overhead of function calls.

If we want to see the coins that make up the optimal solution,

value[0] = 0;

int first[N];

for (int x = 1; x <= n; x++) {

value[x] = INF;

for (auto c : coins) {

if (x-c >= 0 && value[x-c]+1 < value[x]) {

value[x] = value[x-c]+1;

first[x] = c;

}

}

}

and now first contains the optimal coin set.

TC is O(nk) where n is the target sum and k is the number of coins.

* 1. Counting Solutions: Sometimes we need to find the total no. of solutions to a problem. This can also be done with DP. For ex.: For the same {1,3,4} coin problem and a target 5, the total ways are 6

1+1+1+1+1 = 5

1+1+3 = 5

1 + 3 + 1 = 5

3 + 1 + 1 = 5

1 + 4 = 5

4 + 1 = 5

Here the equation is, solve(x) = solve(x-1) + solve(x-3) + solve(x-4)

which is

solve(x) = 0 for x<0

1 for x=0

Sum of all solve(x-c) for x>0

where c is the value of a coin.

And to solve it in c++,

count[0] = 1;

for (int x = 1; x <= n; x++) {

for (auto c : coins) {

if (x-c >= 0) {

count[x] += count[x-c];

count[x] %=m;

}

}

}

where m = 10^9 + 7

we mod x so that the value doesn’t exceed the bounds of int.

* 1. Longest increasing subsequence: In this problem, we have an array {6,2,5,1,7,4} and we have to find the longest subsequence size of elements that are strictly increasing from left to right, which is 3 as {2,5,7} are the longest strictly increasing elements and 3 is their size.

So here we subproblem the longest subsequence, and try to find all the subsequences including the given number at the end then store the size of the longest subsequence for the element and move to the next element. Lastly we pick the largest subsequence size among all the sizes and that’s our solution.

Subsequence size != subsequence ending with an element size

These 2 are different things, here we try to find all subsequences, and we know every subsequence ends with every element. So, we simply look for increasing subsequences ending with elements and then store the max subsequence size. So for ex.:

subsequence ending with element 7 at index 4,

there’s 2,5,7 , 1,7 , 5,7 , 6,7 , 2,7 and the max size subsequence is 2,5,7 which gives us 3. But if we wanted to know max subsequence size ending at element 1 at index 3, it’d be just 1, however max subsequence size till element 1 at index 3 is 2.

So, let’s say the function to find the longest subsequence is length(x), it returns the size of the longest subsequence ending at element x.

length(0) = 1

length(1) = 1

length(2) = 2

length(3) = 1

length(4) = 3

length(5) = 2

So we can say,

the base case is size 1,

we look for elements at index i<k where array[i] <array[k] and then if true, length(k)= max(length(k), length(i)+1). We add 1 as 1 is the base case.

In C++, it is

for (int k = 0; k < n; k++) {

length[k] = 1;

for (int i = 0; i < k; i++) {

if (array[i] < array[k]) {

length[k] = max(length[k],length[i]+1);

}

}

}

then the max subsequence size is simply length.max();

TC is O(n2)   
There’s a very specific solution with nlogn time factor but it is less intuitive.