Bit Hacks

Remove lowest set bit in x:
$x = x \,\&\, (x-1)$ — 100111010 → 100111000

Isolate lowest set bit in x:
$x = x \,\&\, \sim(x-1)$ — 100111010 → 000000010

Parity: 1 if the number of ones is odd, otherwise 0

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

$2^{16} = 65536$
$0xFFFF = 65535 = \underbrace{1111.1111.1111.1111_2}_{16 \text{ ones}}$

Extract ith bit of x:
$x \,\&\, (1 \ll i)$

Swap Bits: Only needed if the bits are not the same. If needed, it's like flipping bits:
$x \;\hat{}= ((1 \ll i) \mid (1 \ll j))$

Reverse Bits:
Use precomputed lookup table for 16bit words.
If $x = y_1 y_2 y_3 y_4$, then $rev(x) = rev(y_4) rev(y_3) rev(y_2) rev(y_1)$

Closest integer with same number of ones:
- If we flip bits at index k1 and k2 with k1 < k2, the absolute difference is $2^{k1} - 2^{k2}$
  - ↳ swap the 2 rightmost consecutive bits that differ

Error Handling: (#include <stdexcept>)
throw std::invalid_argument("x is not zero");

x·y without arithmetical operators
- add using bitwise operations
- multiply using shift-and-add
  - ↳ grade-school algorithm

x/y using only +, −, and bitshifts
Idea: Find largest k s.t. $2^k \cdot y \le x$, subtract 2^k from x, and add 2^k to the quotient

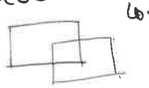
x^y
↳ Idea: $\left(x^{\frac{y}{2}}\right)^2$ or $x \cdot \left(x^{\frac{y}{2}}\right)^2$ ...
↳ If y negative: replace x by $\frac{1}{x}$ and y by −y

Reverse digits:
Get last digit of integer x:
$x \,\%\, 10$

Check if decimal is palindrome:
↳ negative number can't be palindrome
↳ number of digits in integer x:
$n = \lfloor \log_{10} x \rfloor + 1$
least significant digit:
$x \,\%\, 10$
most significant digit:
$x / 10^{(\lfloor \log_{10} x \rfloor + 1)}$
remove most significant digit:
$x \,\%= pow(10, num\_digits - 1)$
remove least significant digit:
$x /= 10$

Generate uniform random numbers in range [a, b], given random 0/1 values:
↳ same as in range [0, b−a]
↳ easy if $b - a = 2^i - 1$
  ↳ generate each bit of i-bit-number
↳ otherwise, search smallest i s.t. $2^i - 1 \ge b - a$, and redo generation if out of range

Rectangle Intersection (Parallel to X- and Y-axis)
↳ Focus on when rectangles don't intersect
  ↳ Handle x- and y-dimension separately



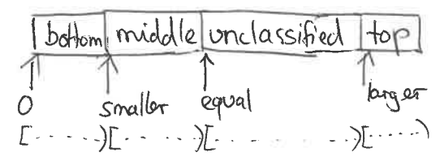
```
binary_search(A.begin(), A.end(), 42):
  returns true if 42 is in A, where A is sorted in increasing order
lower_bound(A.begin(), A.end(), 42):
  returns pointer to first position of an entry in sorted A which is ≥ 42
Get actual index by doing
  lower_bound(A.begin(), A.end(), 42) - A.begin()
auto it = lower_bound(A.begin(), A.end(), 42);
if (*it == 42) {
  std::cout << *it << "found at index" << std::distance(A.cbegin(), it) << "\n";
}
upper_bound(A.begin(), A.end(), 42):
  returns pointer to next higher number than 42
std::fill(A.begin(), A.end(), 42)
std::swap(x, y)
std::min_element(A.begin(), A.end()) => return iterator to element!!!
std::max_element(A.begin(), A.end())
std::reverse(A.begin(), A.end())
std::rotate(A.begin(), A.begin()+shift, A.end())
  ↳ rotate left by 2: rotate(A.begin(), A.begin()+2, A.end())
  ↳ rotate right by 2: rotate(A.begin(), A.begin()+A.size()-2, A.end())
std::sort(A.begin(), A.end())
```

Custom comparator ascending sort:

```
std::sort(A.begin(), A.end(), [](const auto& lhs, const auto& rhs) {
    return lhs.key < rhs.key;
});
```

Dutch-National-Flag: (<, =, >)

typedef enum {RED, WHITE, BLUE} Color;



in the beginning, smaller = equal = 0 and larger = A.size()
while (equal < larger) // keep iterating as long as there is an unclassified element
  if A[equal] < pivot
    swap(A[smaller++], A[equal++])
  else if A[equal] == pivot
    equal++;
  else
    swap(A[equal], A[--larger])

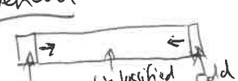
Increment arbitrary precision integer: -> Grade-school algorithm

```
++A.back()
for (int i = A.size()-1; i > 0 && A[i] == 10; --i) {
    A[i] = 0;
    ++A[i-1]
}
if (A[0] == 10) {
    A.insert(A.begin(), 1);
}
return A;
```

Arrays I

Often easy brute-force with extra storage, trick is to do it in-place instead

Even/Odd: while not Even < not Odd ...




```
#include<algorithm>
std::swap(x,y)
```

like push_back, but slightly faster:
v.emplace_back(12);

```
std::vector<int> v = {1,2,3};
std::vector<int> v(12,0); // 12 entries each is 0
std::array<int,2> a = {1,2};
int a = v.front() // direct reference to first entry
v.back() // direct reference to last entry
```

v.insert(v.begin(), 1);
// insert 1 at the beginning of v

Subarray: std::vector<int> sub(A.begin()+i, A.begin()+j) // sets it to A[i...j-1]

Arrays II

Multiply arbitrary-precision integers:

↳ use grade-school algorithm!
↳ product has at most n+m digits for n- and m-digit operands

Example: 123 · 987

```
  861
 984 .
1107 ..
121401
```

remove ~~leading~~ zeros:
result = { find_if_not(begin(result), end(result), [](int a){return a==0;}), end(result)};

```
#include<algorithm>
std::find_if_not(A.begin(), A.end(), [](int i){return i==0;});
```

↳ returns iterator to the first element in A which is not 0
↳ if no such element is found, returns A.end()

```
std::find_if(A.begin(), A.end(), [](int i){return i!=0;});
```

Advance through array: Given array where each position stores how far you can go right, return whether you can reach the end from the start

↳ keep track of the furthest index we can advance to
furthest_reach_so_far = max(furthest_reach_so_far, max_advance_steps[i]+i)
↳ starts at zero

Delete duplicates from sorted array:

↳ sorted array: repeated elements appear one after another

```
DeleteDuplicates(vector<int>* A_ptr){
  vector<int>& A = *A_ptr;
  //...
}
```

↳ simple, just keep track of current write_index
↳ if (A[write_index-1] != A[i]) A[write_index++] = A[i]

Buy and sell stock once:

↳ keep track of min_price_so_far #include<limits>
~~↳ minimum~~ std::numeric_limits<double>::max()

Buy and sell stock twice: (second buy must be made after first sale)

↳ forward phase: ~~best~~ store best solutions for A[0...j], j∈[1,n-1]
↳ backward phase: store best solution for A[j...n-1], j∈[1,...,n-1]
↳ Combine results from forward and backward search

Enumerate all primes to n:

prime ≙ >1 and only divisible by 1 and itself

↳ Sieve of Eratosthenes

~~bool is_prime[n+1]~~

```
std::deque<bool> is_prime(n+1, true)
```

↳ sieve p's multiples:

```
for (int k=p*p; k<=n; ~~k++~~ k+=p)
  is_prime[k] = false
```

↳ all numbers of form k·p where k<p have already been sieved out
↳ further optimization: skip even numbers, start from 2i+3 with i=0

```
static_cast<long>(j)
```

↳ need to use long because p² might overflow

Permute elements of an array: (apply permutation)

↳ every permutation can be represented by a collection of independent permutations, each of which is cyclic, that is, moves all elements by a fixed offset, wrapping around.
⇒ identify disjoint cycles
↳ to find cycle including i we keep going forward (from i to P[i]), until we get back to i

↳ use sign-bit in permutation array as ~~addi~~ storage for bools

Cyclic permutation:

↳ Perform each cyclic permutation one-by-one.

Compute next permutation:

↳ look at entry just before longest decreasing suffix
↳ swap that entry with the smallest entry in the suffix that is larger than e
↳ then, sort the entries in the suffix from smallest to largest — reversing the new suffix is enough since suffix is already decreasing

```
return empty vector:
vector<int> getEmpty(){
  return {};
}
```

```
find_if(vec.rbegin(), vec.rend(), [&](int a){return a>vec[k];});
```

↳ capture by reference
↳ search from the back

Sample offline data

Random subset of size k

```
#include<random>
std::default_random_engine seed((std::random_device())());
int v = uniform_int_distribution<int>{from, to}(seed);
```

↳ Select random entry in A[0...n-1], ~~place it at A~~
swap it with A[0].
↳ Select random entry in A[1...n-1], swap it with A[1]...

Sample online data (sample of size k from input stream)

↳ read first k entries into array
↳ The (n+1)th packet should belong to the new subset with probability k/(n+1)
↳ choose one of the packets uniformly to remove, if new entry is selected randomly
↳ Reservoir Sampling
• Generate random number in [0, num_seen_so_far-1], and if this number is in [0, k-1], we replace that element from the sample with x

```
#include<sstream>
std::stringstream s(myString);
std::string s = S.str();
S.str(newString) //sets the string of s to newString
```

std::istringstream → only reading
std::ostringstream → only writing
We can use it like with std::cout and std::cin

Compute random permutation:

```
#include<vector>
#include<numeric>
std::vector<int> perm(n);
std::iota(perm.begin(), perm.end(), 0);
```

↳ fills perm with increasing ~~elements~~ numbers, starting from 0 // 0,1,2,3,4,...,n-1

↳ Sample random entry in A[i...n-1], swap it with A[i]

Compute random subset of size k:

$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$

↳ simulate A with a hash table H
↳ if i in H, then its value is stored at A[i] in the brute-force-algo
↳ if i not in H, it implicitly implies A[i]=i

```
std::unordered_map<int, int> m;
auto ptr = m.find(42)
// ptr->second is the value
// ptr->first is the key
// if not found, ptr is m.end()
```

Use emplace_back instead of push_back in vector!

Arrays III

Generate nonuniform random numbers:
(with probabilities $p_0, p_1, \dots p_{n-1}$)

↳ prefix sum of probabilities
↳ random number between [0, 1)
↳ find correct interval in prefix sum array using binary search

```
#include<numeric>
std::accumulate(first,last,sum,myfun)
```
initial value of the sum — replace addition by smth else, e.g. `int myfun(int x,int y){ return x-y;}`

↳ returns the sum of all values lying in a range between [first, last) with the variable sum

```
std::partial_sum(first,last,b,myfun)
```
b: array where to write the resulting prefix sum to

↳ prefix sum, elements to be added lie in [first, last)

example with vector: (std::vector<double> prefix_sum)
`std::partial_sum(probs.begin(), probs.end(), std::back_inserter(prefix_sum));`

Uniform random double in [0, 1):
```
std::default_random_engine seed((std::random_device())());
double d=std::generate_canonical<double,numeric_limits<double>::digits>(seed);
```

↳ `std::distance(prefix_sums.cbegin(), upper_bound(prefix_sums.cbegin(),prefix_sums.cend(),d))-1`

Sudoku-checker:
↳ straightforward, use `std::deque<bool>` instead of `vector<bool>`
↳ use hasDuplicate (const vector<vector<int>>& field, int start_row, int end_row, int start_col, int end_col)

Spiral Ordering of 2D array:
↳ like an onion!

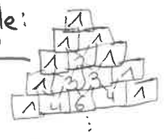
→ First n-1 elements of
- first row
- last col
- last row reverse
- first col reverse

↳ keep an offset value
↳ we can keep track of already processed items by setting them to zero

Rotate 2D array:
↳ layer by layer

↳ or just change the indexing
$A[i][j] \to A[n-j-1][i]$

Rows in Pascal's Triangle:



↳ keep arrays left-aligned
↳ ith entry in jth row is 1 if $j = 0$ or $j = i$, otherwise it is sum of (j-1)st and jth entry in the (i-1)th row

$$\sum_{i=1}^{n} i = \frac{n \cdot (n-1)}{2} \in O(n^2)$$

Interconvert strings and integers
- 1 digit at the time
- be aware of negative sign, and of 0
- least significant digit of integer: $x \% 10$
- remaining digits of integer: $x / 10$
- add digits from the back
- reverse result string at the end
- base-10-number $d_2 d_1 d_0 = 10^2 \cdot d_2 + 10^1 \cdot d_1 + 10^0 \cdot d_0$
- begin from leftmost digit and with each succeeding digit, multiply the partial result by 10 and add that digit

```
return {s.rbegin(), s.rend()} //return reversed string
```

digit to char:
`c = '0' + digit`

char to digit: (if char is in 'A'...'F':)
`digit = char - 'A' + 10`
`digit = char - '0'`

Base conversion:
Convert string representing integer in base b_1 to base b_2
↳ first convert into decimal int

$\log(a^b) = b \cdot \log(a)$

```
#include<cctype>
int isdigit(9) → returns nonzero integer
isdigit(A) → returns zero
```

Spreadsheet column encoding:
- There are 26 chars in ['A'..'Z']

↳ base-26-number to integer (except that A corresponds to 1, not 0)

```
res=res*26+c-'A'+1
return res
```

Reverse all words in sentence:
"Alice likes Bob" → "Bob likes Alice"
↳ First, reverse entire string
↳ Second, reverse each word in the string

```
#include<algorithm>
std::reverse(s.begin(),s.end())
```

`s.find(needle, pos)` → returns first position of needle in s (after and including pos)
Or `std::string::npos`, if not present

Compute all mnemonics for phone number:
↳ return all possible character sequences corresponding to number
(don't have to be legal words)
↳ use recursion
↳ emplace_back instead of push_back!

const array<string, 3> mapping = {"ABC", "DEF", "G"}...

Look-and-say-problem:
↳ return n-th integer (as string) in look-and-say pattern
↳ iteratively apply the rule n-1 times

`std::to_string(123)` → returns "123"

Convert Roman to Decimal:
↳ start from the right, check if I[s[i]] < I[s[i+1]]

```
#include<unordered_map>
std::unordered_map<char,int> I={{'I',1},{'C',2},{'X',12}};
```

Compute valid IP addresses:
↳ all possible placements of periods, ☐.☐.☐.☐
all substrings have to be between 0 and 255
↳ space the periods 1 to 3 characters apart
↳ prune when substring is not valid
↳ "00" is not valid → in general, leading zeros are not valid!

`stoi("422")` → returns 422

Write string sinusoidally:
↳ compute indices for first row, second row, third row
↳ 3 passes
s[1] +4→ [5] +4→ [9] ...
s[0] +2→ [2] +4→ [4] ...
s[3] +4→ [7] +4→ [11] ...

Implement run-length-encoding and decoding
eg. aaaabccaa ⟺ 4a1b2c2a
↳ straightforward

```
if(isdigit(c)){ count=count*10+c-'0';}
```

`str.append(42,'A')` → appends 42 'A' characters to the string

Find first occurrence of substring:
Rabin-Karp-Algorithm ($O(m+n)$)
↳ uses hashing
↳ compute hash value of pattern
↳ rolling window for hash in text
↳ only compare potential matches if hashes are equal

$hash(t_{s+1} \dots t_{s+m}) = d \cdot (hash(t_{s+1} \dots t_{s+m-1}) - t_{s+1} \cdot h) \bmod q$

d: # of chars in alphabet; h: d^{m-1}; q: a prime number

(m is length of pattern)

↳ extended ASCII has 256 characters

~~$hash(t_0 \dots t_{m-1})$~~

a b e
$1 + 2 \cdot d^1 + 5 \cdot d^2$
↳ base-d-number }

- encode each m-letter string as a base-d-number, where d = alphabet size
- for avoiding overflow, do this modulo a prime number q

Rolling hash example:

a b e
$1 + 2 \cdot 3^1 + 5 \cdot 3^2 = 52$

b e d
$2 + 5 \cdot 3^1 + 4 \cdot 3^2$
↳ $(52 - 1) / 3 + 4 \cdot 3^2$

`std::nth_element(vec.begin(), v.begin()+n, vec.end())`
↳ rearranges vec s.t. element at v.begin()+n is now the n-th element it would be in the sorted array, left from it are elements ≤, right from it elements ≥

```
bool cmp(int a,int b){
  return a<b;
}
```
~~nth_element(v.begin(),v.begin()+n,v.end(),cmp)~~

Strings
Palindrome Check:
```
for(int i=0,j=s.size()-1; i<j; ++i,--j){
  if(s[i] != s[j]) return false;
}
return true;
```

```
#include<string>
std::string s;
```

`s.append("Gauss");` → adds string to the end (overloaded for +=, operator +=)
`s.push_back('c')` → adds character to the end
`s.pop_back();` → deletes last character
`s.insert(s.begin()+shift,"Gauss");` → inserts before given position (existing text at position shift)
`s.substr(pos,len);` → returns a new string
`s.compare("Gauss");` → returns value in {0, <0, >0}, based on lexicographical ordering
`s.compare(pos,len,str,subpos,sublen);`

s1 == s2 → tests if the string contents are the same

- modify the string itself to save memory

`stoi("42")` → 42
`std::to_string(1/2.0)` → "0.5"

Replace and remove:
- replace 'a' by 'dd'
- delete 'b's
- only first k entries in array need to be processed

- 2 passes/iterations on s
- forward iteration: Get rid of the b's, count number of a's
- We now know total size of resulting string
- Backward iteration: Copy over the new entries, starting from last position

Check Palindrome II:
- alphanumeric character ≙ letter or digit
- ignoring case and non-alphanumeric chars

↳ keep forward idx and backward idx, one pass

```
#include<cctype>
isalnum(c) → checks if c is digit or letter
isalpha(c) → checks if c is letter
isdigit(c) → checks if c is digit
x=tolower(c) → returns c in lowercase
x=toupper(c) → returns c in uppercase
```

Linked Lists

```
#include<memory>
template<typename T>
struct ListNode{
  T data;
  std::shared_ptr<ListNode<T>> next;
};
```

```
//search in list node
while (L&&L->data!=key) {
    L=L->next;
}
return L;
```

(const_shared_ptr, but we can still modify the internals of the object it is pointing to

- don't forget to update next for the head and tail
- often: use 2 iterators, one quicker than the other

std::list → doubly-linked list

std::forward_list → singly-linked list

```
#include<list>
std::list<int> l;
```

```
l.push_front(42); //or l.emplace_front(42)
l.pop_front();
l.front();
l.push_back(42); //or l.emplace_back(42)
```

l.splice(l1.begin(), l2)
↳ inserts/moves all entries from l2 to the beginning of l1, l2 is empty afterwards

l1.splice(l1.begin(), l2, l2.begin())
↳ inserts/moves only first element of l2 to begin of l1

l1.splice(l1.begin(), l2, l2.begin(), l2.end())
↳ inserts/moves elements in [l2.begin(), l2.end) to (l1.begin()

```
l.reverse()
l.sort()
```

```
#include<forward_list>
std::forward_list<int> fl;
fl.push_front(42); //or fl.emplace_front(42)
fl.pop_front()
fl.insert_after(fl.end(),42) //or fl.emplace_after(fl.end(),42)
```
↳ returns iterator to the last inserted element

fl.insert_after(fl.end(), 100, 42)
↳ inserts 100 copies of 42 after fl.end()

fl.insert_after(fl.begin(), myarray.begin(), myarray.end())

fl.erase_after(A.begin())
↳ erases the element after the given iterator, returns iterator to the next element after the one that has been erased

Merge two sorted lists:

- only change next pointers
- take care when one iterator reaches the end

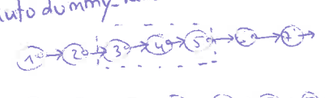
```
shared_ptr<ListNode<int>> dummy_head(new ListNode<int>());
auto tail = dummy_head
```

Reverse a single sublist: [start, end]

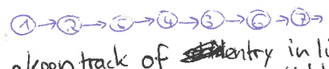
- don't allocate additional nodes
- brute-force: first reverse sublist, then add it back (2 passes)
- can be done in one pass:
  1) Find the predecessor of the node at start
  2) Reverse the sublist

Use dummy_head.

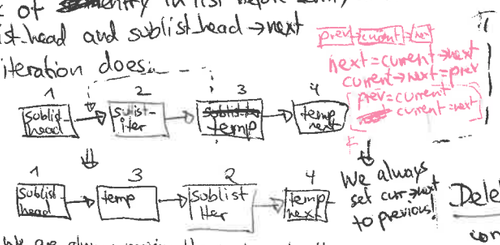
```
auto dummy_head = make_shared<ListNode<int>>(ListNode<int>{0,L});
```



and return dummy_head->next at the end



- keep track of entry in list before entry at start
  sublist_head and sublist_head->next
  - 1 iteration does...



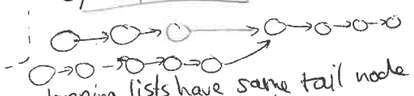
We always set current to previous!

We are always moving the next one to the beginning and updating the next pointers

Test for Cyclicity:

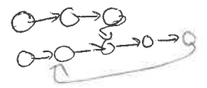
- use one fast and one slow iterator
  - fast: by 2
  - slow: by 1
- has cycle if and only if the two iterators meet
  - from that point, compute cycle length C:
    ~~first node in cycle is the node we end at~~
  - to find first node in cycle, let one iterator by C times next than the other iterator. Then increment both iterators until they meet, the meeting point is the first node in the cycle.
- be careful not computing NULL->next!!!
  - Alternatively: If loop is found, leave fast iterator where it is, set slow iterator to head of the list. Increment both one by one. When they meet, we have found the first node in the cycle.
- If we can change the list nodes, we can also simply store if they have been visited or not...

Test for overlapping cycle free lists



- overlapping lists have same tail node
- compute lengths of both list
- advance iterator on longer list by length difference
- then advance list iterators one by one until they meet.

Test for overlapping lists that may have cycles:



- both lists no cycle: easy
- Only one list cycle: no overlap possible
- both lists cycle: must be the same cycle
  ↳ 2 sub-cases: they meet before cycle (easy), or they reach cycle at different nodes of cycle (do it like in cycle detection)
- If they are both in a cycle, find out if it's the same cycle by going through the cycle in one list and checking

Delete node from singly linked list:

- node is not the last one, we can copy the data field of the successor and then it works in O(1) without needing to find the predecessor
- simply delete the successor, but first copy its data and next entries to the node that had to be originally deleted.

Remove kth-last element from list:

(not knowing the length of the list)

- use 2 iterators, starting from list head
- move first iterator by k
- then move both iterators by 1 until first iterator ends
- second iterator is now pointing to last element, delete its successor (k+1)th

Remove duplicates from sorted list:

- straightforward, update next pointers

Cyclic right shift for singly linked list:

- shift by k
- if k>n, we can shift by k%n
- find tail node
- set successor to tail node to current head
- new head is (n-k)th node in the initial list
- new tail->next becomes nullptr

Even-odd merge:

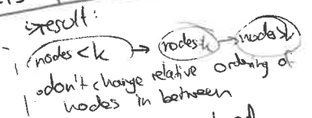
- first all even-numbered entries, then all odd-numbered entries
- make 2 dummy list heads, keep them as tails.
- add to current tail the current node, then switch tails, ...
- append second head to first tail

```
int turn =0;
turn ^=1; //alternate turn
```

Test whether singly linked list is palindromic:

- find middle of the list by having a slow ptr and a fast ptr that moves twice as fast
- reverse second half of list
- Go through first half and reversed second half one-by-one and compare
- If list shouldn't be changed, re-reverse second half of list

List pivoting:



- don't change relative ordering of nodes in between
- build 3 new head nodes

```
shared_ptr<ListNode<int>> less_head(new ListNode<int>());
```

- fill those lists by setting the next pointers correctly
- fix the next nodes of the last entries in these lists
- return less_head->next

Add list-based integers:

- grade-school-algorithm
- make extra list node if there is a carry left
- start with dummy_head

```
shared_ptr<ListNode<int>> dummy_head (new ListNode<int>);
```

- carry = sum/10
- build new ListNode for each digit

```
make_shared<ListNode<int>>(ListNode<int>{sum%10,nullptr})
```

return dummy_head->next

Stacks

Stack: Last in, first out
↳ very good for reverse iterators

```
#include<stack>
std::stack<int> s;
s.push(42); // puts 42 on top (or s.emplace(42))
int val=s.top(); // says what element is on top (throw exception if stack is empty)
s.pop(); // removes top element, void function
bool e=s.empty(); // says if stack is empty
```

Stack with MAX API:

- for each entry in the stack, store maximum so far when being pushed

```
struct ElementWithMax {
    int element;
    int max;
};
stack<ElementWithMax>...
```

```
#include<stdexcept>
throw std::length_error("stack is empty");
```

- alternatively, to reduce best-case space complexity, keep 2 stacks:
  - 1 for the normal entries
  - 1 for cached MaxWithCount

Evaluate RPN expressions:

Reverse Polish notation
"3,4,+,2,*,1,+" = ((3+4)·(2))+1

- process subexpressions, keeping values in stack
- record partial results

```
#include<sstream>
std::istringstream ss(expr);
std::string token;
const char kDelimiter=',';
while (std::getline(ss, token, kDelimiter)) {
}
```

- if it's a number, emplace stoi(token) to the stack
- if it's an operator, take to top entries from stack, pop them, apply the operator, and emplace the result
- at the end, return stack.top().

```
char c=str.front(); // returns first character in string
```

Test a string over {},(),[] for well-formedness:

- each right parenthesis must match the closest left parenthesis on its left
- put all left-parentheses we see on the stack, as soon as we see a right parenthesis, check if stack.top() matches it, if so do stack.pop(), else return false.

Normalize Pathnames:

- return shortest equivalent pathname
- "/" → current dir
- "../" → parent dir
- traverse input from left to right, splitting on "/"
- put names on stack
- skip "/" ones
- if "../" see, pop from stack

```
throw std::invalid_argument("string is empty");
```

- don't explicitly use stack, instead use vector<string> pathnames, with emplace_back() and pop_back()
- special case if path starts with "//": put it into pathnames-array.
- avoid starting "//"

```
while(getline(ss,token,'/'))...
```

Search a postings list:

Nodes with additional jump-field jumping to another node

- follow jump field if jump target not visited, else follow next search from the next node
  ↳ jump-first-order
- mimic recursion with stack
- order-field initialized with a -1, we update it as we visit a node
- since stack is LIFO, first emplace the next node, then the jump node

(like DFS) iterative implementation

Compute buildings with a sunset view:

- a building does not have a sunset view, if there is a it is to the east of a taller building
- bruteforce: buildings in array, store running maximum height
- improve best-case space complexity by putting buildings on a stack and always popping the now blocked buildings when a new building gets added (all building ≤ new building height are deemed)

```
std::istringstream* sin; int building_height;
while (*sin >> building_height)
```

Queues

Queue: First in, first out
↳ ideal when order needs to be preserved

```
#include<algorithm>
auto it = std::max_element(vec.begin(), vec.end())
//the max element is *it.
```

↳ returns iterator to max element ↳ but this is O(n)

```
#include<queue>
std::queue<int> q;
q.push(42);
int val=q.front(); // first one in the queue
int val2=q.back(); // last one in the queue
q.pop() // remove first one in the queue
```

```
#include<deque>
  ↳ double-ended-queue
std::deque<int> dq;
dq.push_back(123); // or emplace_back(123)
dq.push_front(123); // or emplace_front(123)
dq.pop_back()
dq.pop_front()
int val=dq.front()
int val=dq.back()
```

Binary Tree Nodes in order of increasing depth:

- depth = distance from the root
- level-order-traversal
- use 2 queues
- store nodes of depth i and nodes of depth i+1 in two queues
- a bit like BFS

takes const unique_ptr<BinaryTreeNode<int>>& tree and returns vector<vector<int>> storing the data entries of tree nodes level by level

```
// the beginning, queue<BinaryTreeNode<int>*> curr(depth nodes), next(depth nodes);
// while curr not empty, fill next(depth nodes), filling also this level;
//   then, set new curr(depth nodes) to next(depth nodes).
```

Circular queue:

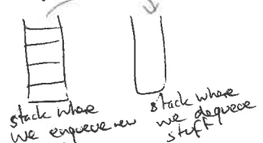
- queue represented by an array, as well as start & end idx.
- dynamically resized (use std::vector)
- track head and tail, as well as #elements
- when resize is necessary, first make queue entries to be at 0...#elements-1, then call vector.resize(2·#vector.size())
- use rotate for it, by head elements

```
vector<int> entries;
tail = (tail+1) % entries.size(); // when enqueue
head = (head+1) % entries.size(); // when de-queue
```

```
#include<algorithm>
std::rotate(vec.begin(), vec.begin()+offset, vec.end())
  ↳ rotates left by offset   ↳ element that becomes first element
std::rotate(vec.begin(), vec.begin()+vec.size()-offset, vec.end())
  ↳ rotates right by offset
```

Queue using stacks:

- use 2 stacks one for enqueue, one for dequeue



- whenever dequeue-stack is empty and we want to enqueue, transfer entire enqueue stack to the dequeue stack
- each element is pushed no more than twice and popped no more than twice

emplace()/pop()/top()

Queue with max API:

- updating the current maximum on dequeue (in a bruteforce solution) is slow
- maintain the set of entries that have no later entry greater than them (in the queue) in a separate deque
  - elements in the deque are ordered by the position in the queue, candidate closest to the head of the queue appearing first
  - each element in the deque is greater-or-equal than its successors
  - current max is at the head of the deque
  - On dequeueing, if the removed element is at the head, pop it from the deque; otherwise deque remains unchanged
  - when enqueueing, iteratively delete from queue's tail elements smaller than new element, then add new element to queue's tail.
- alternatively, represent the queue by two stackWithMax stacks, one for enqueue, one for dequeue

Binary Trees

- useful for representing hierarchy
- depth of a node is the distance to the root
- height of a tree is the highest depth of its nodes

```
template<typename T>
struct BinaryTreeNode {
    T data;
    unique_ptr<BinaryTreeNode<T>> left,right;
};
//sometimes, there is also a
    weak_ptr<BinaryTreeNode<T>> parent;
      ↳ takes a shared_ptr in its constructor
```

full binary tree: each internal node has 2 children

perfect binary tree: each internal node has 2 children and all leaves are at the same depth

complete binary tree: every level, except perhaps the last, is completely filled, and all nodes are as far left as possible

Number of internal nodes in full binary tree: #leaves − 1.

Perfect binary tree of height h contains $2^{h+1}-1$ nodes, of which 2^h are leaves

complete binary tree on n nodes has height $\lfloor \log_2 n \rfloor$

log(x) → log base e
log2(x) → log base 2
log10(x) → log base 10

left (right) skewed tree: Always only left (right) child

Inorder traversal: (left subtree, parent, right subtree

preorder traversal: root, left subtree, right subtree

postorder traversal: left subtree, right subtree, root

Space complexity of recursive algorithms: don't forget size of the call stack!

Tree algorithms often get a `const unique_ptr<BinaryTreeNode<int>>& root` as argument!
↳ we can test for not null by `if(root)`

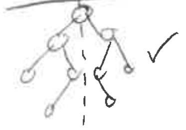
Test if Binary Tree is balanced:

- height-balanced, if for each node, height of subtrees differs by at most 1
- do post-order traversal (with early stop), returning a (if one of subtrees not balanced)

```
struct BalancedStatusWithHeight {
    bool balanced;
    int height; //contains subtree height if balanced
};
if (tree==nullptr) return {true,-1};
abs(x), max(x,y) ... return {isbalanced,height};
```

Binary Trees II

Test if binary tree is symmetric:



Recursive definition of symmetry!

- symmetric, if tree is nullptr
  - otherwise, check recursively with tree→left and tree→right

bool symmetric(const unique_ptr t1, t2):
- not symmetric, if t1→data ≠ t2→data, or if only one of t1, t2 is nullptr
- otherwise, we need to check if symmetric(t1→left, t2→right) and symmetric(t1→right, t2→left)

Lowest Common Ancestor in Binary Tree (no parents)

lca: Node furthest from the root that is ancestor of both nodes
- nodes don't have parent field
- recursive postorder traversal

Return
```
struct Status{
  int num_target_nodes; //0,1,2 depending on how many of the node1, node2 are present in the tree
  BinaryTreeNode<int>* ancestor; // if both are present in the tree, here goes the LCA
};
```

if (tree == nullptr) return {0, nullptr}
- try with tree→left and tree→right, if one of them has num_target_nodes == 2 return their status result
- new num_target_nodes is (left res. num_target_nodes + right res. num_target_nodes + (act Node == node 1) + (act Node == node 2))
- return {num_target_nodes, num_target_nodes == 2 ? tree.get() : nullptr}

Lowest Common Ancestor in Binary Tree (with parents):

- problem is easy if both nodes are the same distance from root,
- first, get depth of both nodes
- let the deeper node go up until it is at the same depth as shallow node
- then, let move go up by 1 at the same time, until both reach the same node → that's the lca

int depth_diff = abs(depth0 - depth1);

for getDepth, we use const BinaryTreeNode<int>* node, because we're going to change that pointer. Call it with node0.get(), where node0 is of type const std::unique_ptr<BinaryTreeNode<int>>&

Sum of binary root-to-leaf paths where each node stores either 0 or 1:

- integer for the path from the root to any node equals integer for parent node ·2 + bit at the node
- if node is leaf return its integer, else return the sum of the results from left and right children
- start by calling it with (tree_root, 0) partial path sum
- since we start at the root and give the value at the root so far to its children, we don't need parent pointers!

Root to leaf path with specified sum:

- each node labeled with an integer
  - path weight: sum of node weights on the path to the root
  - question: is there a leaf with given path weight?
- recursive algo, give path weight of parent
  - return true if found
  - if (node == nullptr) return false
  - for leaf nodes, return whether its path weight equals the target path weight, else, return whether call for left or for right child returned true

argument is of type const std::unique_ptr<BinaryTreeNode<int>>&

Inorder Traversal without recursion:

- nodes don't have parent pointers
- simulate function call stack

```
stack<const BSTNode<int>*> s;
const auto* curr = tree.get()
vector<int> result;
while(!s.empty() || curr){
  if (curr) {
    s.push(curr)
    //Going left
    curr = curr->left.get()
  } else {
    //going up
    curr = s.top();
    s.pop();
    result.emplace_back(curr->data)
    //going right
    curr = curr->right.get();
  }
}
```

Preorder Traversal without recursion:

- nodes don't have parent pointers
- preorder: last in, first out order
- use stack of nodes
- put root on stack
- while stack not empty:
  - take top of stack and pop it, print it.
  - Then first push right child, then left child to stack.

kth node in an inorder traversal:

- where each node stores the size of its subtree
- If the left subtree has L nodes, then the kth node in the inorder traversal is the (k-L)th node if we skip this subtree
  - if k ≤ L, the desired node lies in the left subtree

```
--- while iter != nullptr
if (left_size + 1 < k)
  k -= (left_size + 1)
  iter = iter->right.get()
else if (left_size == k-1) return iter;
else iter = iter->left.get();
```

Compute successor in inorder traversal:

- each node stores its parent
- study the node's right subtree
- Case analysis
  1. If node has right child, Successor is leftmost node in the right subtree.
  2. If node has no right child and is left child of its parent, Successor is the right child of its parent
  3. If node has no right child and it's right child of its parent, go up until parent != nullptr and we are a left child
     - then, it's the parent
- can be root with it's parent being nullptr if we are at the end of the inorder traversal

Inorder traversal with O(1) space: (or we can call always successor)

- nodes have parent fields
- how to tell if node is left or right child of its parent?
- record the subtree's root before we move to the parent
- compare subtree's root with parent's left child
- keep pointer to prev and curr
  - curr starts as tree.get(), prev starts as nullptr

```
while (curr != nullptr) {
  BinaryTreeNode<int>* next;
  if (curr->parent == prev) {
    //we came down to curr from prev
    if (curr->left) //keep going left
      next = curr->left.get();
    else {
      result.emplace_back(curr->data);
      //done with left, so going right if not empty, else going up
    }
  } else if (curr->left.get() == prev) {
    //we came to curr from its left child
    result.emplace_back(curr->data);
    //done with left, so go right if right is not empty, else go up
  } else {
    //done with both children, so go up
    next = curr->parent;
  }
  prev = curr;
  curr = next;
}
```

Reconstruct Binary Tree from traversal data!

- Given preorder and either preorder or postorder traversal, we can reconstruct binary tree
- here, inorder and preorder traversal are given.
- assume each node has a unique key
- focus on the root
- root node is first node in preorder traversal
  - allows us to split inorder traversal into inorder traversal for left subtree, root, inorder for right subtree.
  - this tells us how many nodes are in left subtree
  - subsequence of k nodes after the root is the preorder traversal for the left subtree
- recursive algorithm, naively in O(n²) for skewed tree because of O(n) finding root in inorder traversal seq
- speedup by using hashmap mapping each key of node to its position in inorder traversal

#include<unordered_map>

unordered_map<int, size_t> ...

m.emplace(inorder[i], i)

m.at(preorder[preorder_start])

std::make_unique<Node<int>>(Node<int>{data, left, right});

- build subtree recursively with preorder[pstart : pend-1] and inorder[istart : iend-1]

Don't forget size of the function call stack!

Reconstruct Binary Tree from Preorder Traversal with markers:

- marks when left or right child is empty
- don't examine from left to right
- recursive solution, where the call for left subtree modifies subtree_idx (given as pointer), ++(*subtree_idx)
- if preorder[*subtree_idx] == nullptr return nullptr
- auto left subtree (preorder, subtree_idx_ptr)
- auto right subtree (preorder, subtree_idx_ptr)

return make_unique<Node<int>> (Node<int>{*subtree_key, move(left_subtree), move(right_subtree)})

std::move

//returns unique_ptr<Node<int>>

Form linked list from leaves of binary tree:

- build list incrementally

list<const unique_ptr<Node<int>>*> leaves;

- recursive function

```
if (tree != nullptr)
  //if we are a leaf:
    leaves.emplace_back(&tree);
  //else:
    //first do left subtree, then right subtree
    leaves.splice(leaves.end(), CreateListLeaves(tree->left));
    leaves.splice(leaves.end(), CreateListLeaves(tree->right));
return leaves;
```

Compute exterior of binary tree:

- Handle root's left child and right child in mirror fashion
- path to leftmost leaf is going left if a left child exists, otherwise going right
- compute ↙ in one traversal, ↖ in another, then append ↙ and ↖ to the root node ptr

returning list<const unique_ptr<Node<int>>*>

- res.emplace_back(&root)
- res.splice(res.end(), LeftAndLeaves(tree→left(true)) → preorder trav
- res.splice(res.end(), Right(tree→right, true)) → postorder trav
- return res

bool is_boundary

- emplace_back, if is leaf or is Boundary, new is boundary true, if we are left or only child
- first call it recursively on the left descendent, emplace back if we are leaf or boundary
- result.splice(result.end(), Right(tree→left, is_boundary && tree→right == nullptr))

Compute right sibling tree:

- each node has extra level-next field
- input is perfect binary tree
- do level-order traversal
- when at level i, set next fields for nodes in level i+1
- record starting node for each level
  - will be needed for the next iteration

Locking in Binary Tree:

- a node cannot be set to lock if any of it's descendants or ancestors are in lock
- we have parent pointers
- Track number of locked nodes in subtree in additional node field
- single-threaded
- keep lock counts for ancestors on path to root updated when locking and unlocking

for(auto iter=parent; iter != nullptr; iter = iter→parent)

Heaps

- complete binary tree
- heap property: key at node ≥ keys in subtree (max-heap)
- can be implemented as an array, children of node i are 2i+1 and 2i+2
- O(lg n) insertion, O(1) getMax(), O(lg n) deleteMax(), O(n) search arbitrary key
- deletion: replace root with last leaf and bubble down
- insertion: add new leaf and bubble up
- good for k longest strings question
  - use min_heap keeping the k longest strings seen so far, if new string is larger than min replace it
- good if you only care about largest or smallest elements

#include<priority_queue> → this is a max-heap

priority_queue<int> pq; pq.emplace(12)

pq.push(12) or pq.emplace(12)

int val = pq.top(); → throws exception on empty stack

pq.pop()

pq.empty()

min-heap in C++:

priority_queue<int, vector<int>, function<bool(int, int)>> min_heap ([](const int& a, const int& b) { return a >= b; });

Merge sorted files:

- use min_heap
- we can directly put the iterators into the heap and have a custom compare function

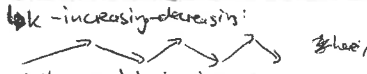
```
struct IteratorCurrAndEnd {
  bool operator>(const IteratorCurrAndEnd& other) const {
    return *current > *other.current;
  }
  vector<int>::const_iterator current;
  vector<int>::const_iterator end;
};
```

priority_queue<IteratorCurrAndEnd, vector<IteratorCurrAndEnd>, std::greater<>> min_heap;

IteratorCurrAndEnd{arr.cbegin(), arr.cend()}

Sort Increasing-Decreasing Array:

k-increasing-decreasing:



here, k=6

- like combining k sorted arrays
- Instead of reversing subarrays, we can use
  - start iterator: A.cbegin() + A.size() - 1 - i,
  - end iterator: A.crbegin() + A.size() - start_idx

typedef enum {INCREASING, DECREASING} SubarrayType;

- use merge-sorted-arrays from before

Sort almost sorted array:

- each number is at most k away from its correctly sorted position (it is k-sorted)
- after we have read k+1 numbers, the smallest number in the group must be smaller than all following numbers
- we need to store k+1 numbers and be able to extract minimum number and add a new number → use min_heap!

priority_queue<int, vector<int>, std::greater<>> min_heap

Takes istringstream* sequence as input

int x;

for (int i=0; i<k && *sequence >> x; ++i)

Heaps II

Compute k closest stars:

- use max-heap containing closest k stars so far
- if new star is closer than max-heap.top(), do max-heap.pop() and max_heap.emplace(new star)
- euclidean distance: $d(x,y) = \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2}$

stod("12.5") → gives 12.5 as double value

string line;
getline(mystream, line, ',') → gets the next entry, using ',' as split delimiter

- at the end, fill closest_stars vector with the k remaining stars in the max-heap, and return {closestStars.rbegin(), closestStars.rend()}

Median of online data:

- keep running median
- avoid looking at all values each time you read a new value
- median of a collection divides each collection into 2 equal parts
- when a new element is being added, the parts change by at most 1 element, and the element to be moved is the largest of the smaller half or the smallest of the larger half
- Use max-heap for smaller half and min-heap for larger half, keep the heaps balanced in size
- min-heap (for larger elements) is allowed to be 1 element larger than max-heap (for smaller elements)
- median is average of the top heap elements if they are the same size, otherwise the top of the minheap

input is istringstream* sequence

```
int x;
while (*sequence >> x)
priority_queue<int, vector<int>, greater<>> min_heap;
priority_queue<int, vector<int>, less<>> max_heap;
```

k largest elements in max heap

- given max heap represented as array, don't modify the heap
- use partial-order property of max heap (parent ≥ its children)
- keep max-heap to track which index to process next, initialized holding index 0 (which represents A[0])
- always extract-max from that extra heap, then emplace 2i+1 and 2i+2 into that extra heap, if they are < A.size()

```
struct HeapEntry{
  int index, value;
};
priority_queue<HeapEntry, vector<HeapEntry>, function<bool(HeapEntry, HeapEntry)>>
  pmh([](const HE& a, const HE& b){ return a.value < b.value; });
```

Stack API using heap:

- store an additional value with each element that is inserted
- track the insertion order by a global "timestamp" for each element, which we increment on each insert
  → use this timestamp to order elements in a max heap

```
struct ValueWithRank{
  int value, rank;
  bool operator<(const ValueWithRank& other) const {
    return rank < other.rank;
  }
};
...
max_heap.emplace(ValueWithRank{x, timestamp++});
```

Searching

Binary Search:

```
int bsearch(int t, const vector<int>& A) {
  int L = 0, U = A.size() - 1;
  while (L <= U) {
    int M = L + (U - L)/2; // to avoid overflow
    if (A[M] < t) {
      L = M+1;
    } else if (A[M] == t) {
      return M;
    } else {
      U = M-1;
    }
  }
  return -1;
}
```

→ This returns index of any occurrence!

```
const static function<bool(const Student&, const Student&)> CompGPA = [](const Student& a, const Student& b) {
  return a.grade < b.grade;
}
binary_search(students.begin(), students.end(), target, CompGPA);
```

#include <algorithm>

std::find(A.begin(), A.end(), target)
→ returns iterator to first occurrence of target, if found, else A.end()

std::binary_search(A.begin(), A.end(), target)
→ returns true if target is in sorted array A

std::lower_bound(A.begin(), A.end(), target)
→ returns iterator to first element greater-or-equal than target in A, where A is sorted

std::upper_bound(A.begin(), A.end(), target)
→ returns iterator to first element greater than target in A, where A is sorted

Search sorted array A for first occurrence of target in A:

- don't stop when you first seek, instead, set upper to mid-1 (as if it were larger than target) and store mid in result variable, but don't return yet.
- (in the beginning, result is -1)

Search sorted array for element equal its index:

- array of distinct integers
- reduce to ordinary binary search
- if A[j] > j, then no entry after j can satisfy the condition, because the array is sorted
- if A[j] < j, then no entry before j can satisfy the condition
- slightly simpler: search for entry where A[i] - i == 0

Search cyclically sorted array for its min:

- sorted array that has been rotated / shifted an unknown number of times
- assume all elements are distinct
- use divide&conquer principle
- if A[m] > A[n-1], then min must lie in [m+1, n-1] (right)
- if A[m] < A[n-1], then min must lie in [0, m] (left)
  → binary search

Compute Integer Square root:

- takes non-negative integer, returns largest integer whose square is less-or-equal to the integer k given
- look out for corner case
- not allowed to use sqrt() function
- binary search, start from [0,k]
- if $x^2 \le k$, no number smaller than x can be the result
- if $x^2 > k$, no number greater than x can be the result
- algo terminates when left > right, result is then left-1, because every number less than left has a square ≤ k and left's square is greater than k.

Compute real square root:

std::numeric_limits<double>::epsilon()
→ tolerance to use when comparing doubles, diff must be ≤ epsilon

- if x<1, square root can be larger than x, e.g. $\sqrt{\frac{1}{4}} = \frac{1}{2}$. → start from [x, 1.0]
- else, start from [0,x]
- keep searching as long right-left < tolerance
- at the end, return left.

Search in 2D sorted array:

- nondecreasing rows and nondecreasing columns
- check if a number exists in array
- eliminate a row or column per comparison
- if x < A[0][0] then no row or col can contain x
- look at external cases (corners)
- compare with A[0][n-1]
  → if x == A[0][n-1], return true
  → if x > A[0][n-1], x is greater than all elements in row 0
  → if x < A[0][n-1], x is smaller than all elements in column n-1.
- start from top-right corner
- update row and col after each elimination

Find min and max simultaneously:

- minimize total number of comparisons
- a<b and b<c implies a<c
- partition the array into min candidates and max candidates by comparing successive pairs
  ⇒ Gives n/2 candidates for min and n/2 candidates for max
- n/2-1 comp. to find min from candidates; " " " max " "
- total #comparisons: $3 \cdot \frac{n}{2} - 2$
- Can implement it in a streaming fashion to reduce additional memory

#include<algorithm>
std::minmax(a, b)
→ returns std::pair<int,int> p with p.first = min(a,b) and p.second = max(a,b)
→ It contains references! We need to give values to it!!!

- pair<int,int> act_minmax = std::minmax(A[0], A[1]);
- always update the min when we have a min candidate, and the max when we have a max candidate
- If there is an odd number of elements in A, don't forget to compare with the last one!
- return a struct MinMax { int min, max; }; for more elegance.

Find the kth-largest element:

- distinct entries
- use divide&conquer with randomization
- do it in-place without completely sorting
  → QuickSelect algorithm
- select random pivot, partition into greater than pivot and smaller than pivot
  - if there are k-1 elements larger than the pivot, then the pivot is the k-th largest element
  - if there are more than k-1 elements larger than the pivot, we can discard the elements ≤ pivot
  - if there are less than k-1 elements larger than the pivot, we can discard elements ≥ pivot

Generate random integer in [left, right]:

```
#include <random>
using namespace std;
default_random_engine gen((random_device())());
int r = uniform_int_distribution<int>{left, right}(gen);
```

Find the missing IP address:

- given large file of IP-addresses each is 32 bit integer (8bit.8bit.8bit.8bit)
- find IP-adress which is not in the file
- use as much hard drive space as wanted, minimize RAM usage
- Can you be sure there is an address which is not in the file?
- largest possible IP-address is 255.255.255.255
  → we cannot always search the largest entry and add 1 to it, as this could overflow (but it's a good heuristic)

hash table requires ~10 byte per integer overhead space

- use bit-array-representation for the set of all possible IP-addresses (2^{32} ones)
  → first pass: set to 1 for each encountered address
  → second pass: iterate through all possible addresses until we find one set to zero
  → This still uses too much memory
- make multiple passes through the input file
- count #of IP-adresses whose leading bit is 0 and #of IP-adresses whose leading bit is 1
  → 32 passes
- we can reduce the #of passes by focusing on groups of bits

Go back to beginning of input file stream:

```
ifs.clear();
ifs.seekg(0, ios::beg);
```

Find the duplicate and missing elements:

- If array contains n-1 distinct elements from [0,n-1], we can find the missing element by computing $\left(\sum_{i=0}^{n-1} i\right) - \left(\sum_{i=0}^{n-2} A[i]\right) = \frac{(n-1)\cdot n}{2} - \sum_{i=0}^{n-2} A[i]$
- If array contains n+1 elements, each within [0,n-1], with exactly one duplicate element, it is $\sum_{i=0}^{n} A[i] - \frac{(n-1 \cdot n)}{2}$

Can also be done via XOR:

(XOR(0,1,...,n-1)) XOR (XOR(A[0]...A[n-2])) gives the missing number
→ because every element in the array cancels out with an element in the set {0,1,...,n-1}, leaving only the missing element.

(XOR(A[0],...,A[n])) XOR (XOR(0,1,...,n-1)) gives the duplicate number, because it is the only one not cancelling out.

Given array with n-1 elements from [0,n-1] where 1 entry is duplicated and 1 entry is missing, find the duplicated and the missing entry:

- multiple passes through the array
- ~~let t be the element appearing twice and m be the missing number, then~~
  ~~$\sum_{i=0}^{n-2} A[i] = \frac{(n-1)n}{2} + t - m$~~ → need a second equation to solve.

(XOR(0,1,...,n-1)) XOR (XOR(A[0],...,A[n-2])) = t XOR m

- since m≠t, there must be a bit in m XOR t that is set to 1, i.e. m and t differ in that bit.
  → The ones in XOR are exactly where the bits differ
- suppose t and m differ in kth bit
  → compute XOR of all numbers in [0...n-1] where k-th bit is set to 1
  → compute XOR of all array entries where kth bit is set to 1
  → XOR those two, let it be h.
  → h is either t or m
  → find out which one by doing one pass through the array and checking if it is the duplicate or missing element

x&(~(x-1)) isolates the least-significant bit in x

^ is XOR-operator

Hash Tables

- ↳ don't update keys in hashtables, instead, remove key, update it, add it back
- insert, lookup, and delete in amortized O(1)
- good hash function: fast to compute, spreads elements well. plus we always require that equal keys have equal hash
- rolling hash for string: easy to recompute if we move window by one
- example for string rolling hash:

```
const int kMult = 997;
int val = 0;
const int modulus = // some large prime number
for (char c : str) {
    val = (val * kMult + c) % modulus;
}
return val;
```

- ↳ good data structure to represent a dictionary, i.e. a set of strings
  ↳ but sometimes a trie may be better
- one could use sorted strings as keys for anagrams.
- in unordered_map<string, vector<string>>, no need to extra check if key was not present before! We can directly call my map[key].emplace_back("first")...

Get unsigned char from string:
unsigned char c = static_cast<unsigned char>(str[i])

Create unordered_set from a list
std::unordered_set<std::string> s (myList.begin(), myList.end());

std::hash<std::string> (myString)
↳ returns a size_t value with the string's hash code

- hash function for a set of strings: we can simply XOR them.
  hash<string>(s1) ^ hash<string>(s2) ^ ... ^ hash<string>(sn)
- we can cache some hashes for performance...

If we want to hash a class, it needs to implement
bool operator==(const MyClass& other) ...
And we need a struct for the Hash function:

```
struct MyHash {
    size_t operator()(const MyClass& a) {
        // return the hash of a
    }
};
```

And then we do unordered_set<MyClass, MyHash> s

```
#include <unordered_set>
unordered_set<int> s;
s.insert(42); // or s.emplace(42);
s.erase(42);
s.find(42);
s.size();
```

↳ returns pair of iterator and boolean (was newly added?)
↳ iterator to element with given key
↳ returns iterator to the element if found, or s.end() else.

```
#include <unordered_map>
unordered_map<int, string> m;
m.insert({42, "Gauss"}) // or emplace({42, "Gauss"})
m.erase(42)
m.find(42)
m.size()
```

↳ returns pair of iterator and boolean
if the bool is false, the element was not updated!

```
#include <functional>
hash<int>(12);
hash<string>("Hallo")
hash<unique_ptr<...>>
```

↳ provides hash functions for basic classes in C++

Test for Palindromic Permutations:

- Palindrome: String that reads the same forwards and backwards
- test whether letters forming a string can be permuted to form a palindrome
  ↳ at most one character is allowed to appear an odd number of times
- store character frequencies in a hash map

```
size_t odd_freq_count = 0;
return none_of(begin(freqs), end(freqs), [&odd_freq_count](const auto& p) {
    return (p.second % 2) && ++odd_freq_count > 1;
});
```

Implement an ISBN-cache:

- string of length 10, first 9 are digits, last char is a check character (sum modulo 11, with 10 replaced by X)
- Create Cache for looking up book prices by their ISBNs.
- Use Least Recently Used (LRU) policy for cache eviction
- Lookup and insert update entry to most recently used
- amortize cost of deletion or use auxiliary data structure
- store ISBN as key and pair of price, most recent lookup time
  (when cache is full, needs O(n) for finding which element to evict
  ↳ thus, use Queue as auxiliary data structure:
- in the hash table, store for each ISBN key its location in the queue and its price
- when an entry is looked up and found, move it to the front of the queue (requires using a linked list for the queue, s.t. items in the middle of the queue can be moved to the head)
- when an entry is added and the cache is now too large, remove element at the queue's tail from both the cache and the queue

(the queue elements are the ISBN numbers)

```
typedef unordered_map<int, pair<list<int>::iterator, int>> Table;
Table my_Table;
list<int> my_queue

void MoveToFront(int isbn, const Table::iterator& it) {
    my_queue.erase(it->second.first);
    my_queue.emplace_front(isbn);
    it->second.first = my_queue.begin();
```

↳ Front element is at begin() of the list!

Is anonymous letter constructible?:

- can cut out single chars
- count # of distinct chars appearing in the letter
- single pass over the letter, storing the char count in a hash table
- then single pass over the magazine, decrementing char count in the hash table if the key is present → if new count is zero, delete entry.
- return true if hashmap is now empty.

Compute LCA, optimizing for close ancestors:

- nodes have parent pointers
- time complexity depending on the nodes distance to the LCA
- alternate moving upwards and store nodes already visited in a hash table
- each time we visit a node, check if it has been seen before

unordered_set<const Node<int>*> s;

```
auto iter0 = node0.get() // we get
auto iter1 = node1.get() // const unique_ptr<Node<int>>& as arguments...
```

S.emplace(iter0)
↳ returns false, if was already in the set

Compute k most frequent queries:

- Compute k strings that appear most frequently in an array
- use hash map for string frequencies
- maintain a min-heap of the k most frequent strings
- add first k strings to hash table
- when new string is added, compare its frequency with the freq in the min-heap. If larger than freq of min heap root, delete min heap root and add new string to the minheap
- at the end, elements in the heap are the solution
- reduce runtime complexity further by doing quickselect on array of unique strings, using their frequencies as values to compare to...

Find nearest repeated entries in array:

- find distance between closest pair of entries in array
- use hash table, storing for each entry seen so far when it was last seen, use this for updating the current min distance
- In the beginning, set
  int best = std::numeric_limits<int>::max()

Find smallest subarray covering all values:

- take array of strings and set of strings Q, return indices of starting and ending index of a shortest subarray containing all words in the set
- keep track of latest occurrences of query keywords as we process A
- use doubly-linked list L to store last occurence (index) of each keyword in Q, and hash table H to match each keyword in Q to corresponding node in list L
- each time a word in Q is encountered, remove its node from L (which we find by using H), create new node which records the current index in A, and append the new node to the end of L; also update H.
  ↳ each keyword in L is ordered by its order in A
  ↳ If L has |Q| words, the new best candidate equals current index minus index of first node in L

list<int> loc;
unordered_map<string, list<int>::iterator> dict
↳ in the beginning, do
emplace(s, loc.end())
for ∀s ∈ Q

Find smallest subarray sequentially covering all values:

- given array of strings A and array of strings B, find shortest subarray in A that covers all elements in B in correct order
- elements in B are distinct
- for each index in A, compute shortest subarray ending at that index which fulfills the specification
- use hash table to map keywords to their most recent occurrences in A as we iterate over A
- use second hash table mapping each keyword to the length of the shortest subarray ending at the most recent occurrence of the keyword, covering all keywords before it
- the 2 hash tables give us the ability to determine the shortest subarray sequentially covering the first k keywords given the shortest subarray sequentially covering the first k-1 keywords
- If current string in A[i] is jth keyword, update most recent occurence of the keyword to i
- Shortest subarray ending at most recent occurrence of first j-1 keywords + elements from most recent occurrence of (j-1)th keyword to i make the shortest subarray ending at most recent occurrence of first j keywords.
- We can map each keyword to its index in B and then use vectors instead of unordered_maps for the rest

vec.back()
↳ same as vec[vec.size()-1]

- dist_to_prev_keyword = i - latest_occurrence[keyword_idx-1]
- shortest_subarr_length[keyword_idx] = distance_to_prev_keyword + shortest_subarr_length[keyword_idx-1]

Longest subarray with distinct entries:

- Given array, return length of longest subarray with distinct entries
- What to do if an array satisfies property from i to j, but from i to j+1 does not?
- use hash table storing the most recent occurrence of each element
- keep track of longest duplicate free subarray ending at current element
- check if A[i] appeared before in current longest good subarray so, newest best subarray starts at this earlier occurrence + 1

```
auto dup_idx = most_recent_occ.emplace(A[i], i)
if (!dup_idx.second) { // was already there
    if (dup_idx.first->second ≥ longest_dup_free_subarr_start_idx) {
        // still the old entry
        res = max(res, i - longest_dup_free_subarray_start_idx);
    }
    dup_idx.first->second = i;
}
return max(result, A.size() - longest_dup_free_subarray_start_idx);
```

Find length of longest contained interval:

- find largest subset of integers in the array s.t. all elements within interval are present
- we don't need total ordering
- initialize unordered_set to store the unprocessed items in A, it will store s (A.begin(), A.end())
  unordered_set<int> s
- iterate over A, whenever A[i] is in the unordered_set, search for largest interval containing A[i] and all its neighbors, remove elements from unordered_set after querying them to avoid recomputation

unordered_set<int> s;
s.count(42) // returns either 1 if 42 is present, else 0.
s.erase(42) // removes 42, returns either 1 if 42 was present, else 0

Average of top 3 scores:

- Generalize to computing top k scores
- use min-heap tracking the top 3 scores for each student
- use unordered_map<string, priority_queue<int, vector<int>, greater<>>> student_scores;
- extract elements from heap using top() and pop()

Compute all string decompositions:

- Given string sentence and array of strings words, return starting indices of all substrings which are concatenations of all words in words (each word must appear exactly once)
- assume all words have equal length n
  ↳ then, only one distinct word in words can be a prefix of a given string
- use unordered_map<string, int> storing the frequency of each word in words
- for each index i in A as possible start candidate, only words from words set, and no word more or less than in words_to_freq (overall), then use unordered_map<string, int> curr_string_to_freq.

```
#include <string>
string s;
s.substr(start_pos, length)
```

Test Collatz conjecture:

- come up with heuristics
- test for overflow: if number that should be larger is suddenly ≥
- hypothesis can fail if we return to previous number in seq, implying it will loop forever; or if it tends to ∞
- Reuse results for smaller numbers

then std::overflow_error("...")

Hash function for chess

- 6 classes of pieces, or empty, each field can be each piece, black or white
  ↳ 4 bits per square
  ↳ 64 board
  ↳ 256 bits represent state of board
- wanted: rolling hash
- XOR is associative, commutative, and fast to compute. Also, a ^ a = 0.
- we could treat each state as a number and do $h(board) = \sum c_i p^i$ with prime
- generate and store random code for each possible state of each field
- then, XOR them.

Sorting

- Heapsort is in-place but not stable
- Mergesort is stable but not in-place
- Quicksort has $O(n^2)$ worst-case runtime
- Insertion Sort: Best for ≤10 elements in array
  - ↳ like when adding a new card to a sorted hand

```
#include<algorithm>
std::sort(vec.begin(), vec.end(),
    [](const Student& a, const Student& b){
    return a.age < b.age; });
```

or

```
struct Student {
    bool operator< (const Student& other) const {
        return age < other.age;
    }
    int age;
}
```

and then `std::sort(vec.begin(), vec.end());`

- To sort an array, use std::sort
- To sort a list, use std::list::sort

Intersection of sorted arrays:

- Given 2 sorted arrays, return new array containing elements that are present in both arrays
- Input arrays can have duplicates, but returned array should be duplicate-free
- If one array is much smaller than the other, do binary search on the larger array (using std::binary_search(B.begin(), B.end(), A[i])
- If arrays have similar length, iterate through both at once (increment aPos if A[aPos]<B[bPos], bPos if A[aPos]>B[bPos])

```
while (aPos < A.size() && bPos < B.size())
    if (A[aPos] == B[bPos] && (aPos==0 || A[aPos] != A[aPos-1]) ..
    ⋮
```

Merge two sorted arrays:

- Given 2 sorted arrays A and B, with A having enough space for {A, B}, merge B into A
- Avoid repeatedly moving entries
- Fill the larger array from its end

```
while (a>=0 && b>=0)
    A[write_idx--] = A[a] > B[b] ? A[a--] : B[b--]
while (b ≥ 0)
    A[write_idx--] = B[b]
```

no need to also check for a≥0, because if so here, then they are already at correct places

Remove first-name duplicates:

- O(n) time + space by using hash table with custom hash and equals

```
auto hash = [](const Node& n){return 13*n.a + 37*n.b;};
auto equal = [](const Node& n1, const Node& n2){return (n1.a==n2.a)&&(n1.b==n2.b);};
std::unordered_map<Node, int, decltype(hash), decltype(equal)> m (10, hash, equal)
```

(10 ↑ minimum # of buckets)

- O(n log n) time and O(1) space by first sorting and then erasing duplicates

```
std::sort(vec.begin(), vec.end(), [](const Entry& a, const Entry& b){return a.firstName < b.firstName;});
vec.erase(unique(vec.begin(), vec.end()), vec.end());
```

→ unique() removes adjacent duplicates and returns iterator to the element that follows the last element not removed. erase() which vec to the distinct elements.

Render a Calendar:

- Given set of events $[s_i, e_i]$, compute maximum number of events that happen at a same timepoint
- (focus on endpoints)
- Sort the set of endpoints in increasing order (if 2 endpoints have same time, the startpoint come first, if both are same point type, break tie arbitrarily)
- Sweepline over event endpoints
- if current endpoint is a start, increment counter
- if current endpoint is an end, decrement counter
- return max value counter had in this process

Merging Intervals:

- Given set of disjoint closed intervals (with integer endpoints), sorted by increasing order of left endpoint, and interval to be added, return new set of disjoined intervals sorted by left endpoint
- union of closed intervals nonempty (and a new closed interval), if they share a point

1) Add all intervals that end before new interval to be added starts to the result
2) Build union as long as we intersect with current interval, add union when finished with a
3) Add the remaining intervals

```
res.insert(res.end(), intervals.begin() + i,
           intervals.end());
```

↳ inserts all remaining intervals to the end of vector res

use
```
struct Interval {
    int begin, end;
};
```

Union of Intervals:

- integer endpoints
- may be open or closed at endpoints
- compute union of given set of intervals as new set of intervals

(do a case analysis)

- Sort the intervals by the left endpoints, breaking ties by putting closed left endpoint before open left endpoint, otherwise arbitrarily
- Compare most recently added interval (was added to the result) with current interval and check if they intersect
  - ↳ don't intersect: Add to result
  - ↳ do intersect: Update most recently added interval with intersection

Partitioning and Sorting

Array with many repeated entries:

- Rearrange array of students s.t. entries with equal age appear together
- (Count # of students for each age)
- Use unordered_map<int, int> age_to_count
- Use second unordered_map<int, int> age_to_offset, storing start indices for next age in the result array (we will reorder elements from input array to create result array)
  - swap the students to their correct place in the partitioned array
- if we also want the result to be sorted by age, we can use a BST-based map (but BST-insertion takes O(log(n))
  - ↳ counting sort

Team Photo Day-1:

- Given 2 arrays representing heights of team members, return if it possible to permute both arrays s.t. A[i]<B[i] for i=1,...,n
- ↳ Sort the arrays by player heights, then check if constraint is satisfied for all i

Fast sorting for lists:

- stable list sorting
  - ↳ relative order of equal elements must be unchanged
- In lists, inserting elements to correct place doesn't require shifting the rest
- MergeSort for lists can be done in-place
- Find middle of list by using 2 iterators, one moving twice as fast as the other
- implement (recursive) mergesort, maximum function call stack depth will be O(log(n)), since we always halve the lists...

Compute salary threshold:

- put cap on salaries, given salaries
- employees who earned more than cap will be paid cap, others will be paid their normal salary
- target total amount to pay out is given
- sort salaries
- do binary search
- Payroll by cap c:

$$\sum_{i=1}^{n-1} \min(c, A[i])$$

- If cap is too high, no higher cap will work (same with too low)
- as we increase the cap, as long as it does not exceed someone's salary, the payroll increases linearly
  - ↳ Iterate through salaries in increasing order
  - ↳ Compute payloads when cap equals a salary, as well as prefix sum of all salaries before current
  - ↳ will give us that cap needs to lie between kth and (k+1)th salary

$$\sum_{i=0}^{k-1} A[i] + (n-k)\cdot c \stackrel{!}{=} T \Rightarrow c = \left(T - \sum_{i=0}^{k-1} A[i]\right)/(n-k)$$

Binary Search Trees

BST property: it is global!

key(parent) ≥ keys(left_subtree)
key(parent) ≤ keys(right_subtree)

lookup, insertion, deletion: O(log(h)) (↳ tree height)

- Example for height-balanced BST: Red-black tree
  - root is black
  - leaves are black
  - same # of black nodes on each root-leaf path
  - red node has black children
  - nodes are either black or red
- When updating a value in a BST: ① Remove it, ② Update it, ③ Reinsert it

```
template<typename T>
struct BSTNode {
    T data;
    unique_ptr<BSTNode<T>> left, right;
};
```

- In practice, BST uses slightly more space than hash table, but still O(n)

(we often get `const unique_ptr<BSTNode<int>>& tree` as input)

```
#include<set>
std::set<int> s;
//we can iterate using s.begin() and s.end(),
this iterates in sorted order
//for descending order, use
s.rbegin() and s.rend()
```

- *s.begin() → smallest element in BST
- *s.rbegin() → largest element in BST

s.lower_bound(42)
↳ returns iterator to first element ≥ 42

s.upper_bound(42)
↳ returns iterator to first element >42

s.equal_range(42)
↳ returns pair of iterators to upper and lower bound of elements containing the key

(all: O(log(n)))

```
#include<map>
//similar to set...
```

Set with custom comparator:

```
auto cmp = [](int a, int b)
    {return ...};
std::set<int, decltype(cmp)> s(cmp);
std::map<int, int, decltype(cmp)> m(cmp);
```

Iterate over key-value pairs in map:

```
for (auto const& kv : m)
    //use kv.first for key
    and kv.second for value
```

Test if binary tree satisfies BST property:

- max of left subtree ≤ root ≤ min of right subtree
- Go recursively down, → (starting from root) specify allowed range for nodes in subtree
- Or one could do an inorder traversal...
- If property is violated by a node whose depth is small, do a BFS
  - ↳ Use queue storing node as well as lower and upper bound for values in its subtree
  - initialized with (root, -∞, ∞)
  - each time we pop a node, we check its constraint

```
struct QueueEntry {
    const unique_ptr<BinTreeNode<int>>& node
    int lower_bound, upper_bound;
};
// q.emplace(QueueEntry{tree, numeric_limits<int>::min(),
                        numeric_limits<int>::max()});
```

//Get first element of queue q by q.front()

Find first key greater value in BST:

- Given BST and a value, find first node that would appear in BST in an inorder-traversal which is greater than the value

(perform binary search, keeping some additional state)

- search the BST, starting from root
  - ↳ keep track of `BSTNode<int>* first_so_far`
  - ↳ don't go down subtrees which we know are less or equal to value
  - ↳ can be done iteratively with `BSTNode<int>* subtree = tree.get()` and `while (subtree){ }...`

Find k largest elements in BST:

(in decreasing order)

- do a reverse-inorder-traversal
- give `vector<int>* vec` for holding the result, stop as soon as we have visited k nodes
- reverse-inorder-traversal: first visit right subtree, then root, then left subtree

`const unique_ptr<BSTNode<int>>& tree`

Compute LCA in BST:

- assume all keys are distinct
- nodes don't have parent pointers

(take advantage of BST property)

- let a be smaller than b, we want LCA(a,b)
- If both a and b are smaller than root, LCA must lie in left subtree of root
- If both a and b are larger than root, LCA must lie in right subtree of root
- If the root's key is either a or b, root is LCA(a,b)
- If a < root.val, but b > root.val, then root is LCA
- Can be done iteratively (with while loops and auto* p = tree.get())

Reconstruct BST from traversal data:

- distinct keys
- given inorder traversal
- (draw 5 trees on {1,2,3} and their inorder traversal orders)
  - ↳ Impossible from only inorder traversal data
- ↳ Only preorder traversal data:
  - ↳ works. First entry is root, all entries < root are left subtree, all entries larger root are right subtree
- Reconstruct left subtree in same iteration as identifying the nodes which lie in it
- Use constraint that we only want to build subtree on keys less-than / greater-than value at root_idx from preorder-sequence
- Build BST on subtree rooted at root_idx from current preorder-sequence on keys in (lower_bound, upper_bound)
  - ↳ the recursive calls update root_idx
  - ↳ Also works similar if only postorder traversal data is given

Find closest entries in 3 sorted arrays:

- return one entry from each s.t. the minimum interval containing those items is as small as possible
- (How would you proceed if need to pick 3 entries in single sorted array?)
- min interval size = max(a,b,c) − min(a,b,c)
- start with triple containing smallest entries in each array, with s = min(a,b,c) and t = max(a,b,c)
- remove s from triple and bring next element from array it belongs to into triple
  - ↳ this way, we will always have smallest interval starting at min(a,b,c) in the triple
- ↳ find closest entries in k sorted arrays
  - ↳ repeatedly insert, delete, findmin, findmax in collection of k elements → use BST

std::next(it)
↳ returns iterator to the next position

```
std::multimap<int, int> multi;
```

- ↳ each key can have several values
- ↳ each key-value-pair must be unique

```
struct IterTail {
    vector<int>::const_iterator iter, tail;
};
.... IterTail{vec.cbegin(), vec.cend()}
```

Binary Search Trees II

Enumerate numbers of form $a+b\sqrt{2}$:

- a and b nonnegative integers, q integer which is not the square of another integer
  - closed under addition and multiplication
- compute k smallest such numbers
- smallest number is $0+0\sqrt{2}$
  - next candidates are $1+0\sqrt{2}$ and $0+1\sqrt{2}$
  - add both to BST
- repeatedly do extract min and insert (smallest at myset.cbegin())

```
struct ABSqrt2 {
  ABSqrt2(int a, int b) : a(a), b(b), val(a+b*sqrt(2)) {}
  bool operator< (const ABSqrt2& that) {
      return val<other.val);}
  int a, b;
  double val;
}
set<ABSqrt2> candidates;
```

→ O(k log k) time, O(k) space

- Alternative O(n) solution:
  - store result in array A
  - track i: smallest idx s.t. $A[i]+1 > A[n-1]$
  - track j: smallest idx st. $A[j]+\sqrt{2} > A[n-1]$
  - A[n] will be min(A[i]+1, A[j]+√2)
  - increment i, j if they are equal to where the min came from by 1.

Most visited pages problem:

- function to read next line to log file
- function to retrieve the k most visited pages
- (for each page, count # of times it has been visited)
- use std::map for storing visit counts for each page, storing entries are ordered by visit count, ties broken by page id
- when page is added by logfile, retrieve entry in the set, delete it, update it, insert it again
  - find, erase, emplace
- O(log n)
- Alternatively, put each page into array and then use quickselect
- can use hash table to store iterators to the BST

Minimum-Height BST from sorted array

- root is middle, left subtree is [left, middle-1], right subtree is [middle+1, right]
- simple recursive solution
- mid = start + (end-start)/2 //to avoid overflow
- return unique_ptr<BSTNode<int>>
- make_unique<BSTNode<int>>(BSTNode<int>{...});

Insertion and Deletion in BST:

- assume all elements in BST are unique
- (deleting leaves is easy. Pay attention to children of internal nodes when deleting it!)
- minimize # of links to be updated
- Insert by searching for input value (if found, return false) as keys should be distinct
  - update the node whose child was empty subtree according to relative value of the node's key and the input value — parent→left.reset(new TreeNode{keys}) (resets a unique ptr)
- Delete by first finding the node to delete
  - has 0 children: remove child field in the parent of the node to be deleted
  - has 1 child: update parent of node to be deleted to have this child instead
  - has 2 children: replace contents with content of its successor (which must appear in right subchild), and then deleting the successor (easy because it cannot have left child) — minimum of right subtree

Node* raw = unique_ptr_node.release()
- Gives up ownership and returns raw pointer to object

ptr1.reset(ptr2.release())

Test if 3 BST nodes are totally ordered:

- given 2 nodes in a BST and a third node, the "middle", determine if one of the 2 nodes is a proper ancestor (i.e. not the node itself) and the other node is a proper descendant of the middle node
- nodes don't have parent pointers
- all keys are unique
- (For what specific arrangements of the 3 nodes does the check pass?)
- perform searches for the middle from both alternatives in an interleaved fashion
  - If we encounter middle from one node, we subsequently search for the second node from the middle

```
while( search0 != node1.get() &&
       search0 != node3.get() &&
       search1 != node0.get() &&
       search1 != node3.get() ){
  if (search0) {
    search0 = search0->data > middle.data? search0->left.get() : search0->right.get();
  }
  if (search1) similarly...
}
```

- If both searches unsuccessful, or we got from node0 to node1 without seeing middle or from node1 to node0 without seeing middle, we can return false as middle cannot lie between node0 and node1
- then, still search if we can get from middle to the other one of {node0, node1}
- always use BST property which makes us only need to search in 1 subtree!

Range Lookup Problem:

- Find Nearest Neighbor in 2D using 2 BSTs (1 sorted on X-coords, 1 sorted on Y-coords)
- Given BST and an interval, return BST keys lying in that interval
- (How many edges are traversed when the successor function is repeatedly called m times?)
- Use BST property to prune traversal
  - If root holds a key less than left endpoint of interval, no need to traverse left subtree
  - (Similar with larger than right endpoint and right subtree)
  - If root holds key that lies in interval, add it to result
- (passed as a pointer to vector<int>)
- and traverse both subtrees
- Runtime: O(height of BST) for nodes left from interval, O(height of BST) for nodes right from interval, O(interval size) for nodes within interval...

Augmenting BST

- for faster range queries (determining # of keys lying in interval)
- add subtree size for each node
- If duplicates in BST allowed, search for first node that would appear in an inorder traversal
- #keys in [L,U] = n − (#keys < L) − (#keys > U)
  - subtree-size of the root
- updating size field on insert and delete: change size entry for all nodes on path to root
  - O(BST height)

Add Credits:

- design data structure with methods:
  - Insert: add client with credit, replacing any existing entry for the client
  - Remove: delete a specified client
  - Lookup: get # credits for client
  - Add-to-all: Add amount of credits to all
  - Max: return client with highest credit count
- (use additional global state)
- Hash Table has no efficient max
- BST has efficient max, but no global increment
- wrapper!
  - store clients in BST, have wrapper track the total increment amount
- subtract global_increment count from credits for newly added client
- when Lookup, return credits[client] + global_increment
- combine
  - unordered_map<string, int> client_to_credit
  - map<int, unordered_set<string>> credit_to_clients
  - int global_increment
- (if begin() == end(), then the collection is empty)

Recursion

- solution of problems depends partially on solution of smaller problems
  - Identify base cases
  - Ensure progress (that the recursion converges to the solution)
- Divide & conquer: Decompose problem into independent smaller subproblems and solve them

Greatest Common Divisor:

If y > x, then
$gcd(x,y) = gcd(x, y-x)$

$$GCD(x,y) = \begin{cases} x, & \text{if } y == 0 \\ GCD(y, x\%y), & \text{otherwise} \end{cases}$$

- Use recursion as alternative to deeply nested loops
- tail-recursive program can be made iterative by using a while loop, no stack is needed
  - compiler optimizations do that
- If recursive function may end up being called with the same arguments more than once, cache the results → Dynamic Programming

Towers of Hanoi:

- transfer n rings from one peg onto another, third (empty) peg given; never place larger ring above smaller ring
- (if you know how to transfer n−1 top rings, how does it help to move the n-th ring?)
- try examples
- array<stack<int>, 3> pegs
  - initialize pegs[0] to contain all the items
- int from_peg, to_peg, use_peg (as buffer)
- ① Move n−1 pegs from from_peg to use_peg, then biggest entry to to_peg
- ② Move n−1 pegs from use_peg to to_peg, using from_peg as buffer
  - recursive algorithm
  - $O(2^n)$

Generate all non-attacking placements of n queens

- no 2 queens are in same row, column, or diagonal
  - vector<vector<int>> result; Solve(n, 0, make_unique<vector<int>>().get(), &result);
- enumerate all possible placements by backtracking, placing queen in j-th column in j-th row...
  - diff = abs(y_i − y_j) ; if diff == 0 or j − i → collision
- all queens in a diagonal have the same value for row − column
- all queens in an antidiagonal have the same value for row + column
  - if (row == n) res->emplace_back(...)
  - cand->emplace_back(col)
  - if isValid(cand, col) solve(n, row+1, cand, res)
  - cand->pop_back()

Generate Permutations:

- given array of distinct integers, generate all permutations of the array (no permutation may appear more than once)
- (how many possible values are there for the first element?)
- To compute all permutations starting with A[i] we swap A[0] with A[i] and then continue computing permutations on A[1...n−1].
  - restore original state before trying again with A[i] as starting position
  - genPerm(int i, vector<int>* A_ptr, vector<vector<int>>* res_ptr)
  - swap(A[i], A[j])
  - genPerm(i+1, A_ptr, res)
  - swap(A[i], A[j])
- vector<int>& A = *A_ptr
- alternatively, first sort A, then repeatedly call next_permutation(A.begin(), A.end())
  - returns true if we weren't already at the last permutation; also changes A to be its next permutation

```
sort(A.begin(), A.end());
do {
  res.emplace_back(A);
} while (next_permutation(A.begin(), A.end()));
```

- O(n · n!) — time to store each perm. / # perms.

Generate power set:

- Take input set and return set of all subsets of the set, including {} and the set itself
- (There are 2^n subsets for a given set of size n → use a n-bit number!)
- Isolate lowest bit by doing $y = x \& \sim(x-1)$
  - find the index by computing $\log_2(y)$.
- remove lowest bit by doing $x = x \& (x-1)$
- for each subset / index to find the elements

Generate all subsets of size k:

- Compute all size-k subsets of {1, 2, ..., n} with n and k given
- Case analysis
  - 2 possibilities for subset:
    - ① Contains 1
      - Return all k−1 size subsets of {2, ..., n} and add 1 to them
    - ② Does not contain 1:
      - Return all k-size subsets of {2, ..., n}
- Solve(n, k, 1, make_unique<vector<int>>().get(), &result)
- solve(int n, int k, int offset, vector<int>* partial_comb, vector<vector<int>>* res
  - emplace_back(i);
  - Solve(...)
  - pop_back();
- always undo the change in the backtrack loop (before trying next possibility)

Generate string of matched parentheses:

- Given number, returns set of all strings with given number of pairs of matching parentheses
- (think about prefix of string of matched parens)
- build strings incrementally
- ensure that as each additional char is added, the string has the resulting string has the potential to be completed to a string with k matching pairs of parentheses
- string of length < 2k, and we know it can be completed, extend with 1 additional char s.t. result can still be completed
  - add left parens; only works if #left parens we still need is > 0
  - add right parens: it must be that #left parens we need is less than current # of right parens (there have to be unmatched left parens in the string)

```
void solve(int num_left_parens_needed,
           int num_right_parens_needed,
           const string& valid_prefix,
           vector<string>* result)
```

- we can pass valid_prefix+"(" to a function call!!! As well as string constants such as ""!

Generate palindromic decompositions:

- Palindrome: reads the same forwards and backwards
- decomposition of a string is a vector of strings whose concatenation is the string (non-empty)
- (focus on first palindromic string in a palindromic decomposition)
- task is to generate all palindromic decompositions
- enumerate decompositions that begin with a palindrome
  - backtrack (kinda)

```
void solve(const string& input, int offset,
           vector<string>* partial_part,
           vector<vector<string>>* res)
```

- test all possible prefixes for if they are palindromes
  - for(int i=0, j=prefix.size()−1; i<j; ++i, −−j){...} — compare prefix[i] and prefix[j]
- make_unique<vector<string>>().get() → to get the first partition position
- again:
  - partial->emplace_back(prefix)
  - solve(input, i, partial, res)
  - partial->pop_back()
  - undo change when backtracking!

Generate binary trees:

- return all distinct binary trees with a specified number of nodes
- (Can 2 binary trees whose left subtrees differ in size be the same?)
- If left child has k nodes, we should only use right children with n−1−k nodes (because 1 node is the root)
- All binary trees on n nodes:
  - get all left subtrees on i nodes and all right subtrees on n−1−i nodes for i between [0, ..., n−1]
- We need to clone a tree function

```
unique_ptr<Node<int>> clone(
  const unique_ptr<Node<int>>& tree) {
  return tree ? make_unique<Node<int>>
    (Node<int>{0, clone(tree->left), clone(tree->right)})
    : nullptr;
}
```

- vector<unique_ptr<Node<int>>> result...

$$C(n) = \sum_{i=1}^{n} C(n-i)\,C(i-1) = \frac{(2n)!}{n!(n+1)!}$$ Catalan number

Recursion II

Sudoku Solver:

(↳ Apply the constraints to speed up bruteforce algo)
↳ backtracking, checking whether current partial solution is still correct

- we need to only check row, col, and subgrid of newly added entry
- traverse 2D-array 1 entry at a time, if it is empty, try all possibilities for that entry as long as they lead to a valid state...
- Solving Sudoku on N×N grid is NP-complete

Compute Gray Code:

- n-bit Gray code is permutation of $\{0,1,2,\dots,2^n-1\}$ s.t. binary representations of successive integers in the sequence differ in only one place (with wraparound, last and first must also differ in only one place)
- take n as input, return an n-bit Gray code

(↳ write out Gray Code for n=2,3,4)

- build sequence incrementally, adding a value only if it is distinct from all values currently in the sequence, and differs in exactly 1 place with the previous value
  ↳ backtracking; for a starting point, we can also do new unordered_set<int> ({0})

unordered_set has erase(42) and emplace(42) and count(42)

```
bool differsIn1Bit(int x, int y){
  int diff = x ^ y;
  return diff && !(diff & (diff-1));
}
```

- more analytical solution:
  - compute gray code vector for n-1 bits (implicitly begin with 0 at bit index num_bits - 1)

~~Add 1 to set their leading bit to one~~

leading_bit_one = 1 << (num_bits - 1)

- add them in reverse order with leading bit set to 1

Gray code for n-1 bits, MSB=0 | Reversed Gray code for n-1 bits, MSB=1
↳ by doing code[i] | leading_bit_one

Gray code for n bits

Compute the diameter of a tree:

- length of the longest path in the tree

(↳ the longest path may or may not pass through the root)

edges in tree = # vertices - 1

- If longest path passes not through the root, it is max of diameters of its subtrees
- if longest path does pass through the root, it must be between a pair of nodes in subtrees furthest away from the root
  ↳ distance from root to node in the ith subtree T_i that is furthest from it is $f_i = h_i + l_i$, where h_i is the height of T_i and l_i is the length of the edge from the root to the root of T_i
- longest path is the larger of the maximum of the subtree diameters and the sum of the 2 largest f_i's
- base case: tree that has no children → diameter is 0.

```
struct TreeNode{
  struct Edge{
    unique_ptr<TreeNode> root;
    double length;
  };
  vector<Edge> edges;
};
struct HeightAndDiameter{
  double height, diameter;
}
for(const auto& e : r->edges){
```

↳ recursive algo!

Dynamic Programming

- cache results of intermediate computations
  ↳ because same subproblem may appear more than once

Fibonacci-Number:

$$F(n) = \begin{cases} 0, & \text{if } n=0 \\ 1, & \text{if } n=1 \\ F(n-1)+F(n-2), & \text{else} \end{cases}$$

unordered_map<int, int> cache

↳ cache.count(12) returns 1 if 12 in cache and 0 if it is not there

- minimize cache space!
  ↳ often: bottom-up-fashion
  (for example for Fibonacci it is enough to track the last and second last number

Break a problem into subproblems, s.t.

- original problem can be easily solved once subproblems are available
- subproblem solutions are cached

maximum sum of all int-subarrays given array:

~~Given solution for subarray A[0...n-2]~~

- We need to know the subarray amongst all subarrays A[0:i] i<n-1 with the longest sum
  ↳ desired value is S[n-1] - that sum (prefix-sum-array)
- For each index j, the maximum subarray sum ending at j is

$$S[j] - \min_{k \le j} S[k]$$

↳ track minimum S[k] seen so far and compute max subarray sum for each index

; for max subarray sum

Kadane's algorithm:

```
int max_so_far=a[0];
int curr_max=a[0];
for (size_t i=0;i<A.size();++i){
  curr_max=max(a[i],curr_max+a[i]);
  max_so_far = max(max_so_far,curr_max);
}
return max_so_far;
```

Count number of score combinations:

- given final score and scores for individual plays, return number of combinations that result in final score

(↳ count # combs when there are 0 w₂ plays, 1 w₂ plays etc.) — unordered! ↳ each play can appear at most once!

- A[i][j] stores number of score combinations that result in j, using individual scores in $[0,\dots,i-1]$.
  ↳ 1 possibility to result in 0 → A[i][0]=1

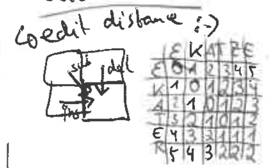
$$A[i][j] = A[i-1][j] + A[i][j-\text{plays}[i]]$$

(only if i-1 ≥ 0; only if j-plays[i] ≥ 0)

vector<vector<int>> v (8, vector<int>(12,0)); — number of entries; initialize the entries with

Compute Levenshtein distance:

(↳ edit distance)



Longest Palindromic Subsequence (LPS):
look at length of LPS of A[i...j]
↳ easy to see how to use smaller subproblems then!

Sometimes it's ... and sometimes ... use this

Count number of ways to traverse 2D array:

- start at top-left corner and go to bottom-right corner
- can only go right or down

(↳ if i>0 and j>0, you can get to (i,j) from (i-1,j) or (i,j-1))

A[0][0]=1 → A[i,j] = A[i-1,j] + A[i,j-1]

- analytical solution:
  each path from (0,0) to (n-1,m-1) takes m-1 horizontal and n-1 vertical steps

$$\binom{n+m-2}{n-1} = \binom{n+m-2}{m-1} = \frac{(n+m-2)!}{(n-1)!\,(m-1)!}$$ steps

possible paths.

Compute binomial coefficients:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$ — O(n·k)

$$\binom{n}{k} = \binom{n}{n-k}$$

$$\binom{n}{0} = \binom{n}{n} = 1$$

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \prod_{i=1}^{k} \frac{n-k+i}{i}$$ — O(k)

Search for sequence in 2D array:

- given 2D array A and 1D array Pattern
  ↳ is there a way to start at some entry in A, go up/down/left/right until exactly the pattern has been visited?
  ↳ Entries in A can be visited multiple times.

(↳ start with length+1 prefixes of the array, move to length 2,3,4... prefixes)

unordered_set<tuple<int,int,int>, HashTuple> cache;

```
struct HashTuple {
  size_t operator()(const tuple<int,int,int>& t) const {
    return hash<int>()(get<0>(t) ^ get<1>(t) * 1021 ^ get<2>(t) * 1048573);
  }
};
```

↳ recursive algorithm with caching...

Knapsack Problem:

- select subset of items that has maximum value and satisfies max total weight constraints
- integer weights and values

(↳ greedy approaches are doomed)

- look at optimal solution if clock is chosen and optimal solution if clock is not chosen

V[i][w] ≙ optimal solution over clocks 0...i-1 and weight w (first i clocks allowed; weight allowed)

$$V[i][w] = \begin{cases} \max\{V[i-1][w], V[i-1][w-w_i]+v_i\}, & \text{if } w_i \le w \\ V[i-1][w], & \text{otherwise} \end{cases}$$

V[n][m] → clocks 0...n-1 are allowed and max allowed weight is m

BedBathAndBeyond.com Problem:

- given dictionary (set of strings) and a name, check whether the name is a concatenation of dictionary words
- dictionary word may appear more than once in a sequence

(↳ determine for each prefix 0...i of the name whether it is a concatenation of dictionary words)

- cache intermediate results, the cache keys being the prefixes of the name
  ↳ corresponding value denotes whether the prefix has a valid word decomposition
- Store the dictionary in a hash table
- A prefix of a given string can be decomposed if it is a dictionary word, or if there is a shorter prefix that can be decomposed and the remainder is a dictionary word

S.substr(pos, len) — position of first char

↳ enough to look back (longestDictWord.size())

Find minimum weight path in triangle:

- Given n arrays of size 1,2,3,...n with weights in their entries, find minimum weight path from top to bottom row
- we can go from (i,j) to (i+1,j) or to (i+1,j+1)
- look at entries in i-th row
  ↳ path that ends at previous row must also be min-weight-path
- prev_row stores min-path sum to each entry in triangle[i-1]

v.swap(v2) ↳ swaps the vectors v and v2 in O(1)

std::min_element (vec.begin(), vec.end()) ↳ returns iterator to smallest element in vec

Pick up coins for maximum gain:

- compute maximum total value for starting player in pick-up-coins game (each player can only take leftmost or rightmost coin at his turn)

(↳ relate best play for first player to best play for second player)

- second player is assumed to play the best move he can

R[a][b] ≙ maximum a player can get when its his turn and there are coins $C_a, C_{a+1}, \dots, C_b$ on the table

- total revenue is constant
- second player will move as to minimize first players revenue

$$R[a][b] = \begin{cases} \max\begin{cases} C[a] + \min\{R(a+2,b), R(a+1,b-1)\}, \\ C[b] + \min\{R(a+1,b-1), R(a,b-2)\} \end{cases} & \text{if } a \le b \\ 0, & \text{otherwise} \end{cases}$$

- cache the results in a vector<vector<int>>...

Count number of moves to climb stairs:

- destination is n steps up
- you can advance 1 to k steps at a time
- compute number of ways to reach destination

(↳ How many ways are there in which you take the last step?)

$$F(n,k) = \sum_{i=1}^{k} F(n-i,k)$$

- cache results in vector<int>

Pretty printing problem:

- messiness of line with b blank characters at its end is b^2
- total messiness is sum of all the lines
- each line has fixed size / max char count
- words on line are separated by 1 blank char
- Given text, i.e. string of words separated by single blanks, decompose text into lines s.t. no word is split across lines and total messiness is minimized

(↳ Focus on last word and last line)

- If optimum placement for i-th word in last line consists of words j, j+1, ..., i, then in this placement, the first j-1 words must be placed optimally

$$M[i] = \min_{j \le i} f(j,i) + M[j-1]$$

(minimum messiness when placing first i words; messiness of a single line containing words j to i inclusive; minimum messiness when placing first j-1 words)

- cache values for M (initialize as vector<int> M(words.size(), numeric_limits<int>::max()))

Find longest nondecreasing subsequence:

- elements in subsequence are not required to immediately follow each other

(↳ Express longest nondecreasing subsequence ending at entry in terms of longest nondecreasing subsequence appearing in the subarray consisting of preceding elements)

- L[i] ≙ length of longest nondecreasing subsequence that ends at i and includes i

$$L[i] = \begin{cases} 1, & \text{if } A[i] \text{ is smaller than all preceding entries} \\ 1 + \max\{L[j] \mid j<i \text{ and } A[j] \le A[i]\} & \text{else} \end{cases}$$

- If we also want the sequence itself, additionally store the index of the last element of the subsequence that we extended to get the value assigned to L[i]

return *max_element(vec.begin(), vec.end());

Master-theorem:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \quad a \ge 1,\ b > 1,\ f(n) > 0\ \forall n > n_0$$

(a: # sub-problems; n/b: size of sub-problem; f(n): time for partitioning/combining)

- Compare f(n) with $n^{\log_b a}$

Case 1: $f(n) \in O(n^{\log_b a - \epsilon})$ for an $\epsilon > 0$
↳ $T(n) \in \Theta(n^{\log_b a})$

Case 2: $f(n) \in \Theta(n^{\log_b a})$
↳ $T(n) \in \Theta(n^{\log_b a} \cdot \log(n))$

Case 3: $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for an $\epsilon > 0$
↳ $T(n) \in \Theta(f(n))$

Greedy Algorithms

- compute solution in steps, each step makes locally optimal solution and it never changes that decision

Compute optimum assignment of tasks:

- each worker does 2 tasks
- tasks are independent
- each task takes fixed amount of time
↳ minimize how long it takes for all tasks to be completed
(• what task should be assigned to the worker who is assigned the longest task?)
- sort the set of task durations
- pair k-th longest with k-th shortest task

```
for (int i=0, j=vec.size()-1; i<j; ++i,--j)
{ ... }
```

- shortest + longest task time is not necessarily total time, see example {1, 8, 9, 10} ...

Schedule to minimize waiting time:

- time a query waits before its turn is called its waiting time
- can do one task at once, each task costs time t_i
- return minimum total waiting time

(• Focus on extreme values)
↳ Serve the short queries first
① Sort queries by service time
② Serve queries by nondecreasing service time.

Interval covering problem:

- given set of closed intervals
- find minimum-sized set of numbers that covers all intervals

(• Think about extremal points)
↳ we can restrict our attention to endpoints without losing optimality
- greedily picking endpoint that covers most intervals might lead to suboptimum results

① Sort all intervals, comparing on right endpoints
② Select the first intervals right endpoint, delete intervals now covered
③ Goto ② as long as there are elements there.

```
std::sort(vec.begin(), vec.end(),
  [](const Foo& f1, const Foo& f2)
  { return f1.val < f2.val; });
```

- we can do without the deletions by checking if current interval start point comes after last visit time...

Invariants

- condition that is true during execution of a program
- Given sorted array and number, determine if there are 2 entries in the array that sum to the number

↳ invariant: maintain subarray that is guaranteed to hold the solution, if it exists
↳ Iteratively shrink from left or right by 1
- if leftmost + rightmost < target, increase leftmost index
- if leftmost + rightmost > target, decrease rightmost index

↳ Work on small examples to hypothesize the invariant

The 3-Sum-Problem:

- given array and number, determine if there are 3 entries (not necessarily distinct) that add up to the number

(• How would you check if a given array entry can be added to two more entries to get the specified number?)
① Sort the array → $O(n \log n)$
② For each array entry A[i] search if there is a 2-sum in A that evaluates to targetSum − A[i]. → $O(n^2)$

Find the majority element:

- given a sequence of strings with one element occurs more than half the time, find the majority element

(• Take advantage of the existence of a majority element to perform elimination)
- have candidate for majority element (initialized to first element), track its count.

↳ when we see equal entry, increment count
↳ when we see different entry, decrement count
↳ If count becomes zero, set next entry to be the candidate
↳ In that case, we throw out 2 elements, where at most 1 was majority element
↳ ratio is still fine

$\frac{m}{n} > \frac{1}{2} \Rightarrow \frac{m}{n-2} > \frac{1}{2}$ and $\frac{(m-1)}{(n-2)} > \frac{1}{2}$

The Gas up Problem:

- cities on circular road, each one has gas
- total gas equals total gas needed to drive in circle
- assuming there is a city we can start from and end at without ever running out of gas (ample point), find it

(• Think about starting with more than enough gas to complete the circuit without gassing up. Track the amount of gas as you perform the circuit, gassing up at each city)
- consider a city where amount of gas in tank is at minimum when we enter that city

↳ does not depend on where we begin from, because graphs are the same up to translation and shifting
↳ This is an ample city

Compute maximum water trapped by a pair of vertical lines:

- given array of possible wall heights, walls placed at distance 1 from their neighbors, find pair i, j of walls where $abs(i-j) \cdot \min(A[i], A[j])$ is maximal.

(• Start with 0 and n−1 and work your way in)
↳ If A[0] < A[n−1], then for any k, the amount of water trapped between 0 and k is less than the amount of water trapped between 0 and n−1 → focus on 1 to n−1
↳ If A[0] > A[n−1], focus on A[0] and A[n−2].
↳ If A[0] == A[n−1], focus on A[1] and A[n−2]
- record most water trapped so far

↳ explore best way in which to trade of width for height

Compute largest rectangle under the skyline:

- Find area of largest rectangle contained in skyline

(• How would you efficiently find the largest rectangle which includes the ith building and has height A[i]?)

- as we advance through buildings, we need to keep track of which buildings have not been blocked yet
- keep an active pillar set when going further right
- Use a stack
- Rightmost building in active pillar set is on top of stack

↳ building below it in the stack tells us how far left the largest rectangle that is supported by the active pillar goes...
① Push i to stack if current height is ≥ height of building on top of stack or if stack is empty (we put indices on the stack, not actual values!)
② While current height < height of building on top of stack, keep removing elements from the stack
↳ While popping from stack, compute areas: popped-height · (popped-idx − curr-idx)
↳ Push not current position, but position from last popped element instead, because that's where the rectangle started
③ Finish of the remaining entries of the stack (synchronized)
④ best to use 2 stacks, one for startpos and one for height!

Graphs

Search a maze:

↳ do BFS or DFS ↳ O(|V|+|E|)

BFS:

```
queue<Node*> q;
q.push(root);
while (!q.empty()) {
  Node* n = q.top(); q.pop();
  visited[n] = true;
  for all neighbors of n:
    if (! visited[n])
      q.push(n);
}
```

↳ DFS is like BFS, but with a stack instead of a queue
↳ DFS can also be implemented recursively)

Paint a boolean matrix:

↳ FloodFill using BFS or DFS from a starting node (as long as color is the same)
↳ no need for extra visited fields as we are flipping the color on a visit

use deque<bool> instead of vector<bool>!

Compute enclosed regions:

- replace all W's that cannot reach the boundary of a grid during flood fill by B

(• It is easier to compute the complement of the desired result)
- Find regions that are fillable with a white path starting from first row, last row, first col, or last col

↳ uses visited[i][j] 2D-array
- Then, make all still unvisited white entries black.

BFS/DFS on a matrix has O(n·m) with n rows and m columns

Deadlock detection:

- "wait-for" graph
- given directed graph, check if the graph contains a cycle

(• focus on "back" edges)
↳ run DFS on the graph, run it from each not yet discovered vertex
- either keep track of which vertices have been visited in current DFS call, or use 3 colors
- back edge is an edge from a node to itself or from a node to one of its ancestors

enum Color {white, gray, black};
(unvisited, visiting, visited — their descendants are still being processed)

- edge to a gray vertex means there is a cycle!
- If there are no white neighbors left to visit, make current node black
- if there is a gray neighbor, we have found a cycle

① Initialize color of all nodes as WHITE
② Do DFS starting at all WHITE vertices
③ in DFS:
① make color of currNode GRAY
② visit all adjacent nodes, if they are WHITE. If there is a GRAY neighbor, we have found a cycle.
③ make color of currNode BLACK

- Visiting a gray vertex means a cycle

Clone a graph:

- each vertex has label and list of pointers to other vertices

(• Maintain a map from vertices in the original graph to their counterparts in the clone)
① Copy root.
② Do BFS on graph, each time we encounter not yet cloned neighbor node, clone it and push it onto the BFS queue (or emplace it)
③ return vertexMap[root]

Making wired connections:

- Find 2-coloring of graph

(• implication of odd-length cycle?)
↳ do BFS with greedy coloring
- If there is an edge from a distance-k-vertex to a distance-k-vertex, we have an odd cycle → no 2-coloring possible

Transform one string to another:

↳ s, t strings, Dictionary (set of strings)
↳ produce t
:⟺ sequence of strings in D starting with s and ending with t, s.th. adjacent strings have same length and differ exactly in one character
↳ model using graph (undirected)
↳ search shortest path from s to t using BFS
- store StringWithDistance in the BFS queue

Team-Photo Day 2:

- find largest # of teams that can be placed on picture simultaneously subject to height constraints

(• form a DAG where paths correspond to valid placements)
- vertices: teams
- edge from u to v iff u can be placed behind v

↳ find longest path in DAG G
↳ do a topological sort of the vertices
- longest path terminating at v is the maximum of the longest paths terminating at v's fan-ins concatenated with v itself

Topological sort:

Perform DFS, adding each vertex to the result after the recursive DFS calls started from it have returned. The result is a stack!

Use:

```
struct GraphVertex{
  vector<GraphVertex*> edges;
  int max_dist = 0;
  bool visited = false;
};
```

To find longest path:

Pop elements from topSort stack one by one, updating
$v \rightarrow max\_dist = \max(v \rightarrow max\_dist, u \rightarrow max\_dist + 1)$ for each neighbor u of v.
↳ also keep track of total max_distance seen so far

Longest path ~~from s to t~~ in DAG:

① Compute topological sort of the DAG using DFS
② For each vertex in topological order:
For each edge (v, w) in the graph:
maxdist[w] = max(maxdist[w], maxdist[v] + weight(v,w))
③ Return the max

Compute shortest path with fewest edges:

- input: Graph G=(V,E), nonnegative edge weights, start vertex s, end vertex t

(• change the edge cost and cast it as instance of the standard shortest path problem)
↳ Modified Dijkstra!
- for each edge cost c, make it (c, 1)

↳ define addition as component-wise addition
↳ compare function compares total cost, and breaks ties on number of edges
↳ Use BST
↳ set<GraphVertex*, Comp> node_set instead of a heap, because we need efficient updates!

```
#include <set>
s.erase(i)
s.emplace(i)
s.begin()
s.cbegin()
```

Time complexity basic Dijkstra: $O((|E|+|V|) \log |V|)$

```
struct GraphVertex{
  struct DistanceWithFewestEdges{
    int dist, min_num_edges;
  };
  DistanceWithFewestEdges d = DistanceWithFewestEdges{numeric_limits::max(), 0};
  struct VertexWithDistance{
    GraphVertex& vertex;
    int distance;
  };
  vector<VertexWithDistances> edges;
  int id;
  const GraphVertex* pred = nullptr;
};
struct Comp{
  bool operator()(const Foo& a, const Foo& b){
    return a->d < b->d; }
};
```