Hacker's Rank

Two sum

1.1	*Max dist sum*	$\max_{n_0<n_1\in[0,N)} x_{n_1}+x_{n_0}+(n_1-n_0)$	*Fidessa*	*maths prop*
1.2	*Max two sum with equal digit-sum*	$\max_{n_0<n_1\in[0,N)} x_{n_0}+x_{n_1} \quad s.t. \quad digitsum(x_{n_0})=digitsum(x_{n_1})$	*Mako Q1*	*dynprog*
1.3	*Max two sum of rooks on chess board*	*two rooks cant attack each other*	*Mako Q2*	*maths prop*
	Max N sum of rooks on chess board	*N rooks cant attack each other*	*Citadel*	*Hungarian algo*
2.1	*2-sum in one or two sorted arrays*	$\min_{n\in[0,N)\ m\in[0,M)} \lvert x_n+y_m-T\rvert$ *where xy are* ***sorted***	*Facebook*	*histogram*
2.2	*K-sum in one unsorted array*	$\min_{n\in[0,N)\ m\in[0,N)} \lvert x_n+x_m-T\rvert$ *where x is* ***unsorted***		*histogram*

Two difference

3.1	*Max profit*	$\max_{n_0<n_1\in[0,N)} x_{n_1}-x_{n_0}$	*alg1.doc*	*dynprog O(n)*
4.1	*Count target profit*	$num_{n_0<n_1\in[0,N)} (n_0,n_1) \quad s.t. \quad x_{n1}-x_{n0}=T$	*Flowtrader*	*dynprog O(n)*
4.2	*Count target absolute profit*	$num_{n_0<n_1\in[0,N)} (n_0,n_1) \quad s.t. \quad \lvert x_{n1}-x_{n0}\rvert=T$		*dynprog O(n)*

Contiguous subsequence

5.1	*Max subseq sum*	$\max_{n_0\le n_1\in[0,N)} \sum_{n=n_0}^{n_1} x_n$	*alg2.doc*	*dynprog O(n)*
5.2	*Max subseq product*	$\max_{n_0\le n_1\in[0,N)} \prod_{n=n_0}^{n_1} x_n$		*dynprog O(n)*
5.3	*Max non-contiguous subseq sum*	$\max_A \sum_{n\in A} x_n \quad s.t. \lvert n-m\rvert>1, \forall n,m\in A$		*dynprog O(n)*
5.4	*Max point puzzle*	*pick number n, get score, remove n±1*	*SMarket Q3*	*dynprog O(n)*
6.1	*Count target subseq sum*	$num_{n_0\le n_1\in[0,N)} (n_0,n_1) \quad s.t. \quad \sum_{n=n_0}^{n_1} x_n=T$	*Maven/FB*	*dynprog O(n)*
6.2	*Count target divisble subseq sum*	$num_{n_0\le n_1\in[0,N)} (n_0,n_1) \quad s.t. \quad \sum_{n=n_0}^{n_1} x_n=0 \bmod T$		*dynprog O(n)*
6.3	*Longest target subseq sum*	$\max_{n_0\le n_1\in[0,N)} n_1-n_0+1 \quad s.t. \quad \sum_{n=n_0}^{n_1} x_n=T$		*dynprog O(n)*
6.4	*Count less-than target subseq sum*	$num_{n_0\le n_1\in[0,N)} (n_0,n_1) \quad s.t. \quad \sum_{n=n_0}^{n_1} x_n<T \quad where \quad x_n\ge 0,\forall n$		*dynprog O(nlogn)*
6.5	*Count less-than target subseq product*	$num_{n_0\le n_1\in[0,N)} (n_0,n_1) \quad s.t. \quad \prod_{n=n_0}^{n_1} x_n\le T \quad where \quad x_n\ge 0,\forall n$	*SMarket Q2*	*dynprog O(nlogn)*

String

7.1	*Longest non duplicated substring*	
7.2	*Longest palindrome substring*	*i.e. mirror image of itself*
7.3	*Longest repeated pattern substring*	*i.e. Tommy BNP interview*

Stack trick

8.1 *Number of stroke / number of function called*
8.2 *Unsorted subseq*
8.3 *Biggest rectangle in histogram*
8.4 *Total size of muddy puddle*

Sorting variant

9.1 *Order statistic*
9.2 *Number of bribed swap*

Other - set / tree / string

10.1 *Set of subsets*
10.2 *Tree layer average*
10.3 *String matching*

Here is the classification of problems. Usually we have :

- `ans` for storing optimum of previous subproblem by comparing all processed `sub`
- `sub` for storing optimum of previous modified subproblem which constrain the subsequence to end at item `n`
- `hist` for counting *(it helps to convert 2D problem into 1D problem)*
- `cum` for sum *(it helps to convert subsequence problem into two-points problem)*

Two-difference	
2.1 *Max profit*	`sub,ans`
3.1 *Count target profit*	`hist,ans`

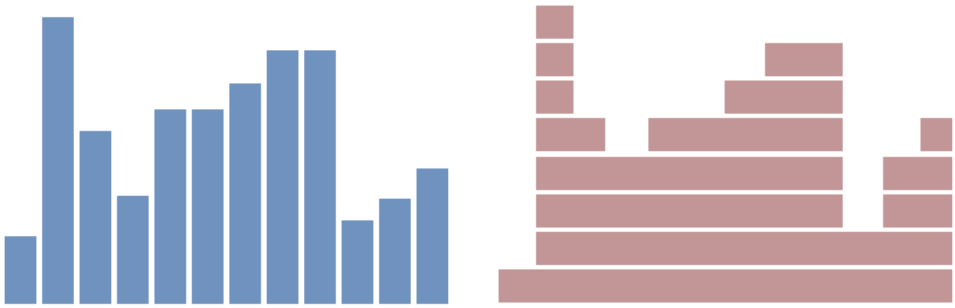
Subsequence	
4.1 *Max subseq sum*	`sub,ans`
4.2 *Max subseq product*	`subx2,ans`
4.3 *Max non-contiguous subseq sum*	`subx2,ans`
4.4 *Max point puzzle*	`subx2,ans`
5.1 *Count target subseq sum*	`cum,hist,ans`
5.2 *Count target divisble subseq sum*	`cum,hist,ans`
5.3 *Longest target subseq sum*	`cum,hist,ans`
5.4 *Count less-than target subseq sum*	`cum,hist,ans`
5.5 *Count less-than target subseq product*	`cum,hist,ans`

	max	*count*
2 pt	`sub #`	`hist`
subseq	`sub #*`	`hist,cum`

remark `#` : provide $O(N^2)$ $O(logN)$ and $O(1)$ solutions
remark `*` : this is covered in Algo2 – Dynamic Prog.doc as well

Stack trick

Whenever we want to convert a vector of integers (blue) into stack (red) like the following, we can make use of the stack trick.



Q1.1 Max distance sum

Please refer to Fidessa document, which makes use of property that arg max f(x) + x > arg max f(x) for monotonic increasing f(x), that decouples double for-loop into 2 single for-loop.

Q1.2 Max two sum with equal digit-sum *Please refer to Mako document.*

Maintain histogram `std::unordered_map<int,<int,int>>`, where digit-sum as the key, and a pair of maximum and second maximum as the value in the histogram. Update the histogram in single linear scan. Very intuitive solution.

```
int sum_with_equal_digit_sum(const std::vector<int>& vec)
{
    std::unordered_map<int,<int,int>> hist;
    for(const auto& x:vec)
    {
        auto iter = hist.find(digit_sum(x));
        if (iter != hist.end())
        {
            if (x > iter->second->first)         {   iter->second.second = iter->second.first;
                                                     iter->second.first  = x;                       }
            else if (x > iter->second->second)   {   iter->second.second = x;                       }
        }
        else hist[digit_sum(x)] = std::make_pair(x, std::numeric_limits<int>::min);
    }

    int output = std::numeric_limits<int>min;
    for(const auto& x:hist) if (output < x.first + x.second) output = x.first + x.second;
    return output;
}
```

We can also rearrange the above so that we have a solution which is more consistent with the solution of other questions in this doc. This time, the histogram stores the maximum value only.

```
int sum_with_equal_digit_sum(const std::vector<int>& vec)
{
    int output = std::numeric_limits<int>min;

    std::unordered_map<int,int> hist;
    for(const auto& x:vec)
    {
        auto iter = hist.find(digit_sum(x));
        if (iter != hist.end())
        {
            if (output < iter->second + x)
                output = iter->second + x;
            if (iter->second < x)
                iter->second = x;
        }
        else hist[digit_sum(x)] = x;
    }
    return output;
}
```

Q1.3 Max two sum on a chess board (Non-attacking rooks) *Please refer to Mako document.*

Given a chess board with score on it, place two non-attacking rooks to get maximum score. The solution makes use of the property that : each rook must be positioned in the maximum (or second maximum) in its row and also in its column. Here is a proof :

- suppose rook *A* is positioned in *(Ay, Ax)* while rook *B* is positioned in *(By, Bx)*
- if rook *B* is not maximum or second maximum in row *By*, then it can always :
- move to another column so that new position *(By, Bx')* is the maximum in the row *By*, if $Bx' \neq Ax$ OR
- move to another column so that new position *(By, Bx")* is 2^{nd} maximum in the row *By*, if $Bx' = Ax$ and $Bx" \neq Ax$
- if rook *B* is not maximum or second maximum in column *Bx*, then it can always :
- move to another row so that new position *(By', Bx)* is the maximum in the column *Bx*, if $By' \neq Ay$ OR
- move to another row so that new position *(By", Bx)* is 2^{nd} maximum in the column *Bx*, if $By' = Ay$ and $By" \neq Ay$

Q2.1 Two sum problem for 1 or 2 sorted arrays

If there is only one sorted array, we can solve it by *Two moving pointer approach* like :

- Citadel interview
- Facebook interview

If there are two sorted arrays, we can solve it by move in *2D* matrix with coordinate `(n,m)` without actually constructing the matrix. There are two tasks : optimization and navigation. Yet, the *O(NlogN)* preprocessing dominates calculation.

```
def target_pair_sum(x, y, target) :
    x.sort() # O(NlogN)
    y.sort() # O(NlogN)
    m = len(x)-1
    n = 0

    optm, optn, min_dist = m, n, abs(x[m]+y[n]-target)
    while True :
        # task 1 : optimization
        if abs(x[m]+y[n]-target) < min_dist : optm, optn, min_dist = m, n, abs(x[m]+y[n]-target)

        # task 2 : navigation
        if x[m]+y[n] < target : n = n + 1
        else : m = m - 1
        if m < 0 : break
        if n > len(y)-1 : break
```

Q2.2 Two sum problem with unsorted array

This method can be extended to 3-sum / 4-sum problem : please read Yubo.doc.

For all K-sum problems, there are two steps :

- scan the input vector and build `hist`
- scan the input vector again and check against `hist` in *O(1)*

Difference between *Q2.2* and *Q4.1* :

- there is no constraint on order of two numbers in *Q2.2*, hence we can build `hist` before starting the linear scan
- there is constraint on order of two numbers in *Q4.1*, hence we can build `hist` on the run within the linear scan
- building `hist` before starting the scan is important for extension to 4-sum, so that we can make it $O(N^2)$

Generalize to K-sum :

- for 2-sum problem, we can put one number `x[n]` in `hist` and scan for another number `x[m]` which takes *O(N)*
- for 4-sum problem, we can put pair sum `x[n]+x[m]` in `hist` and scan for another pair `x[k]+x[l]` which takes $O(N^2)$
- for 3-sum problem, we can either :

- put one number `x[n]` in `hist` and scan for another pair sum `x[m]+x[k]` OR
- put pair sum `x[n]+x[m]` in `hist` and scan for another number `x[k]`
- both are $O(N^2)$

Q3.1 Max profit

It can be done by $O(N^2)$ exhaustive search, $O(NlogN)$ divide and conquer and $O(N)$ dynamic programming. For dynprog, main idea is defining subproblem `n` as maximum profit by buying stock at time `m<n` and selling it exactly at time `n`, so subproblem `n` becomes :

```
suboptimum[n] = max( [buy at m,   sell at n],     // where m < n-1
                     [buy at n-1, sell at n])
              = max( suboptimum[n-1] - vec[n-1] + vec[n],
                     [buy at n-1, sell at n])
```

```
def max_profit(vec) : # O(N^2) exhaustive search
    ans = -float('inf')
    for n in range(len(vec)) :
        for m in range(n+1, len(vec)) :
            if vec[m]-vec[n] > ans : ans = vec[m]-vec[n]
    return ans

def max_profit(vec) : # O(NlogN) divide and conquer
    if len(vec) == 1 : return -float('inf')
    mid = int(len(vec)/2)
    v0  = min(vec[:mid])
    v1  = max(vec[mid:])
    return max(max_profit2(vec[:mid]), max_profit2(vec[mid:]), v1-v0)

def max_profit(vec) : # O(N) dynamic programming
    sub = vec[1]-vec[0]
    ans = sub
    for n in range(2, len(vec)) :
        sub = max(sub+vec[n]-vec[n-1], vec[n]-vec[n-1])
        ans = max(ans, sub)
    return ans
```

Dynprog pattern for $O(N)$

The final implementation shows a common pattern for dynamic programming solution in $O(N)$. For $O(N)$ dynprog, there is one for loop, which keeps updating `ans` and `sub` (or something else, like `cum` or `hist`), `ans` is returned at the end, there is an initialization at the beginning, and that's it. You can find out this pattern by looking at the following examples. Besides, very often, `sub` is defined as the modified problem with constraint that the subsequence ends at index `n`, this idea is used many questions.

Q4.1 Count target profit

Construct `hist` and use it on the run is a useful technique that serves us two purposes :

- save us from constant checking of `m<n`, as `hist` caches information of previous subproblems `m<n` only
- save us from 2D for loop, as it converts one of which into O(1) search in `hist`

```
def count_target_diff(vec, target) : # O(N) dynamic programming
    hist = {}
    ans = 0
    for n in range(len(vec)) :
        # task 1 : O(1) search in hist
        if vec[n]-target in hist : ans = ans + hist[vec[n]-target]

        # task 2 : construct hist on the run
        if vec[n] in hist : hist[vec[n]] = hist[vec[n]] + 1
        else : hist[vec[n]] = 1
    return ans
```

```
# O(N^2) brute force
for m in range(len(vec)) :
    for n in range(m+1, len(vec)) :
        if vec[n]-vec[m] == target : ans = ans + 1
```

Q4.2 Count target absolute profit

There is an extension to the question, what if we count all pairs such that `abs(vec[n]-vec[m]) = target`. The solution becomes :

```
def count_target_diff(vec, target) : # O(N) dynamic programming
    hist = {}
    ans = 0
    for n in range(len(vec)) :
        # task 1 : O(1) search in hist
        if target+vec[n] in hist : ans = ans + hist[target+vec[n]]
        if vec[n]-target in hist : ans = ans + hist[vec[n]-target]

        # task 2 : construct hist on the run
        if vec[n] in hist : hist[vec[n]] = hist[vec[n]] + 1
        else : hist[vec[n]] = 1
    return ans
```

Q5.1 Max subsequence sum

Lets go through all 3 implementations. The dynamic programming here is called **Kadane algorithm**.

```
def max_subseq_sum(vec) : # O(N^2) exhaustive search
    ans = -float('inf')
    for n in range(len(vec)) :
        cum = 0
        for m in range(n, len(vec)) :
            cum = cum + vec[m]
            ans = max(ans, cum)
    return ans

def max_subseq_sum(vec) : # O(NlogN) divide and conquer
    if len(vec) == 1 : return vec[0]
    mid  = int(len(vec)/2)

    cum0 = vec[mid-1]
    ans0 = vec[mid-1]
    for n in range(mid-2,-1,-1) :
        cum0 = cum0 + vec[n]
        ans0 = max(ans0, cum0)
    cum1 = vec[mid]
    ans1 = vec[mid]
    for n in range(mid+1, len(vec)) :
        cum1 = cum1 + vec[n]
        ans1 = max(ans1, cum1)
    return max(max_subseq_sum2(vec[:mid]), max_subseq_sum2(vec[mid:]), ans0 + ans1)

def max_subseq_sum(vec) : # O(N) dynamic programming (Kadane algorithm) # note : no cum is needed !!!
    sub = vec[0]
    ans = sub
    for n in range(1, len(vec)):
        sub = max(sub + vec[n], vec[n])
        ans = max(ans, sub)
    return ans
```

Q5.2 Max subsequence product

The main problem is that if `vec[n]` is negative, previous subproblem maximum will become minimum and vice versa. Thus we need to cache both maximum `sub0` and minimum `sub1` of subproblem in here.

```
def max_subseq_prod(vec):
    sub0 = vec[0]
    sub1 = vec[0]
    ans  = vec[0]
    for n in range(1, len(vec)) : # O(N) dynamic programming
        tmp0 = sub0 * vec[n]
        tmp1 = sub1 * vec[n]
        sub0, sub1 = max(tmp0, tmp1, vec[n]), min(tmp0, tmp1, vec[n])
        ans  = max(ans, sub0) # No need to do : ans = max(ans, sub0, sub1), as sub1 < sub0 is always true
    return ans
```

Q5.3 Max non-contiguous subsequence sum

This time we need two subproblem optimal `sum0` and `sum1`, which stands for the optimal of subproblem of size `n`, with or without the last element included in the sum respectively.

```
def maxSubsetSum(vec):
    sub_wout_n = 0
    sub_with_n = vec[0]
    ans = max(sub_wout_n, sub_with_n)
    for n in range(1,len(vec)) :
        tmp0 = sub_wout_n + vec[n]
        tmp1 = sub_with_n + vec[n]
        sub_wout_n = max(sub_wout_n, sub_with_n)
        sub_with_n = max(tmp0, tmp1, vec[n])
        ans = max(ans, sub_wout_n, sub_with_n)
    return ans
```

Q5.4 Max point puzzle

Please refer to *SMarket* source code. Put numbers into a histogram, then It is like a variation on Q4.3 with updating equations :

```
tmp0 = sub_wout_n
tmp1 = sub_with_n
if jump == 1 : sub_wout_n, sub_with_n = max(tmp0, tmp1), tmp0 + n*hist[n] // Note : we need to pick all numbers at the end
else         : sub_wout_n, sub_with_n = max(tmp0, tmp1), max(tmp0, tmp1) + n*hist[n]
ans = max(ans, sub_wout_n, sub_with_n)
```

Q6.1 Count target subsequence sum

We construct `hist` and apply search on `hist` on the run. Main difference is that we push `cum` into `hist` instead of `vec[n]`.

```
def count_target_subseq_sum1(vec, target) : # O(N) dynamic programming
    hist[0] = 1
    cum = 0
    ans = 0
    for n in range(len(vec)) :
        cum = cum + vec[n]
        # task 1 : O(1) search in hist
        if cum-target in hist : ans = ans + hist[cum-target]
        # task 2 : construct hist on the run
        if cum not in hist :
               hist[cum] = 1
        else : hist[cum] = hist[cum] + 1
    return ans
```

Q6.3 Count target divisible subsequence sum

The key to dynamic programming is to find out things we need to cache for subproblem `n-1` to facilitate the solution to subproblem `n`. For divisibility problem, we just need to store `cum % target` only, as it will cycle in between `[0, target-1]`.

```
def counr_div_by_target_subseq_sum(vec, target) :
    hist[0] = 1
    cum = 0
    ans = 0
    for n in range(len(vec)) :
        cum = cum + vec[n]
        tmp = cum % target
        # merge task 1 & task 2
        if tmp in hist :
            ans = ans + hist[tmp]
            hist[tmp] = hist[tmp] + 1
        else : hist[tmp] = 1
    return ans
```

Q6.3 Longest target subsequence sum

Similar to count-target subsequence sum. Main difference is that `hist[cum]` refering to the number of subsequences with sum `cum`, is replaced by `idx[cum]` refering to the first subsequence with sum `cum`. Everything else are just the same.

```
def longest_target_subseq_sum1(vec, target) : # O(N) dynamic programming
    idx[0] = -1 # this is important
    cum = 0
    ans = 0
    for n in range(len(vec)) :
        cum = cum + vec[n]
        # task 1 : O(1) search in idx
        if cum-target in idx : ans = max(ans, n-idx[cum-target])
        # task 2 : construct idx on the run
        if cum not in idx : idx[cum] = n
        else : pass # record the index on first encounter only
    return ans
```

Q6.4 Count less-than-target subsequence sum (positive number only)

As less-than-target implies multiple targets, we cannot use the `hist` trick, instead we store `idx`, denoting the start of subsequence, which sum is strictly less than target. This method can be applied to sum of positive numbers.

```
def count_less_than_target_subseq_sum(vec, target): # assumption : elements in vec are all positive
    idx[1] = -1 # this is important
    cum = 0
    ans = 0
    for n in range(len(vec)) :
        cum = cum + vec[n]
        # task 1 : O(logN) search in idx
        m = idx.first_element_greater_or_equal_to(cum-target)
        ans = max(ans, n-m)
        # task 2 : construct idx on the run
        if cum not in idx : idx[cum] = n
        else : return ERROR # cum should be increasing as ll numbers are positive
    return ans
```

```
Yes this is asymmetrical ...

Q5.1 is counting with hist
Q5.2 is counting with hist
Q5.3 is max-leng with index
Q5.4 is counting with index
```

Q6.5 Count less-than-target subsequence product (positive number only)

Please refer to *SMarket* source code. This is the same as *Q5.4*, by replacing the following :

```
cum = cum + vec[n]     --->    cum = cum * vec[n]
cum - target           --->    cum / target
idx[0] = -1            --->    idx[1] = -1
```

Q7.1 Longest non duplicated substring

Here `idx[c]` stores the index of the latest encounter of character `c`, (unlike Q9, in which `idx[c]` stores the first encounter).

```
    def longest_non_repeat_string(vec) :
        idx = {}
        sub = 0
        ans = 0
        for n in range(len(vec)) :
            if vec[n] in idx :
                   sub = min(sub+1, n-idx[vec[n]])
            else : sub = sub+1
            ans = max(ans, sub)
            idx[vec[n]] = n
        return ans
```

Q7.2 Longest palindrome

Lets try to solve using dynamic programming, i.e. defining subproblem as palindrome ending at index `n`. However it fails to solve all cases in Leetcode, failure cases are those with cycles, like `ABABABABA` or `ABCBABCBA`.

```
    def longest_palindrome(str): # O(N) dynamic programming
        sub0 = 0 # sub-optimal for even palindrome
        sub1 = 1 # sub-optimal for odd palindrome
        ans  = max(sub0, sub1)
        for n in range(1,len(str)) :
            if str[n] == str[n-sub0-1] :   sub0 = sub0 + 2
            else                       : { sub0 = 0; if (str[n] == str[n-1]) : sub0 = 2 } <--- the trick to make it works for cycle
            ans = max(ans, sub0)
        for n in range(2,len(str)) :
            if str[n] == str[n-sub1-1] :   sub1 = sub1 + 2
            else                       : { sub1 = 1; if (str[n] == str[n-2]) : sub1 = 3 } <--- the trick to make it works for cycle
            ans = max(ans, sub1)
        return ans
```

Adding the pale grey part above can solve the problem, which uses the previous element as a new pivot. Now, consider the *3rd* `B` in `ABCBABCBA`, we cannot find its image when using `C` as the pivot, yet we should not simply reset `sub1` to `1`, since we can find its image when using pevious element as a new pivot.

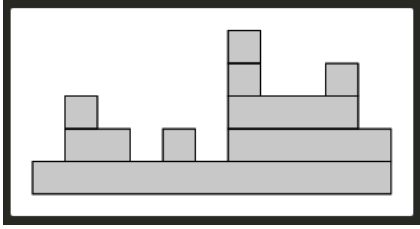
Q7.3 Longest substring with repeated pattern

Given a string and a pattern, find the longest substring, which is a concaternation of repeated pattern.

```
std::string find(const std::string& input, const std::string& pattern)
{
    int n = 0;
    int start = 0;
    int count = 0;
    int max_start = 0;
    int max_count = 0;

    while(n+pattern.size() < input.size())
    {
        auto m = input.find(pattern, n);
        if (m == std::string::npos)
        {
            break;
        }
        else if (m!=n || m==0) // <--- bugfix
        {
            start = m;
            count = 1;
        }
        else
        {
            ++count;
        }

        n = m+pattern.size();
        if (count < max_count)
        {
            max_start = start;
            max_count = std::max(max_count, count);
        }
    }
    return input.substr(max_start, pattern.size()*max_count);
}
```

Q8.1 Number of stroke / number of function called

You would like to paint the skyline using continuous horizontal brushstrokes. Every horizontal stroke is one unit high and arbitrarily wide. The goal is to calculate the minimum number of horizontal strokes needed. For example, the above shape can be painted using nine horizontal strokes.



What is the minimum number of stroke needed?

```
std::uint32_t num_of_stroke(const std::vector<std::uint32_t>& input)
{
    std::stack s;
    std::uint32_t count = input[0];
    s.push(input[0]);

    for(std::uint32_t n=0; n!=input.size(); ++n)
    {
        if (A[n] > s.top())
        {
            count += A[n] - s.top();
            s.push(A[n]);
        }
        else
        {
            while(A[n]<s.top()) s.pop();
        }
    }
}
```

Q8.2 Unsorted subsequence

What is the shortest subsequence, when applied sorting, the whole array becomes sorted? There are 2 answers. Firstly by sorting :

```
def shortest_unsorted_subseq0(nums): # O(NlogN) using sorting
    sorted_nums = list(nums)
    sorted_nums.sort()

    for n in range(len(nums)) :
        if nums[n] != sorted_nums[n] : break
    else : return 0
    for m in range(len(nums)-1,-1,-1) :
        if nums[m] != sorted_nums[m] : break
    else : return 0
    return m-n+1
```

Faster approach is to use a stack, like what we do for biggest rectangle in histogram. Push new index into a stack, keeping popping as the new index is smaller than the top of stack, and record the minimum length of stack. Repeat the same thing backward.

```
def shortest_unsorted_subseq1(nums): # O(N) using stack
    sz0 = len(nums)
    sz1 = len(nums)
    ### forward ###
    s = []
    for n in range(len(nums)) :
        while len(s) > 0 and nums[n] < s[-1] :
            s.pop(-1)
            sz0 = min(sz0, len(s))
        s.append(nums[n]) # in fact, no need to insert anymore after the 1st pop happened
    if sz0 == len(nums) : return 0
    ### backward ###
    s = []
    for n in range(len(nums)-1,-1,-1) :
        while len(s) > 0 and nums[n] > s[-1] :
            s.pop(-1)
            sz1 = min(sz1, len(s))|
        s.append(nums[n]) # in fact, no need to insert anymore after the 1st pop happened
    if sz1 == len(nums) : return 0
    return len(nums)-sz0-sz1
```

Q8.3 Biggest rectangle in a histogram

Given a histogram, find a rectangle inside the histogram having maximum area. Here is the exhaustive implementation in $O(N^2)$.

```
def biggest_rectangle_exhaustive(hist) : // O(N^2) exhaustive search as a benchmark
    max_area = 0
    for n in range(len(hist)) :
        subprob_area   = hist[n]
        subprob_height = hist[n]
        for m in range(n+1, len(hist)) :
            if subprob_height > hist[m] : subprob_height = hist[m]
            if subprob_area < subprob_height * (m-n+1) :
               subprob_area = subprob_height * (m-n+1)
        if max_area < subprob_area :
           max_area = subprob_area
    return max_area
```

Here is the stack-trick implementation in $O(N)$. The idea is :

- suppose `n` and `m` are the begin and end of the rectangle under consideration, where `n < m`
- there is no need to fully scan all `n in range(len(hist))` and all `m in range(n+1,len(hist))`
- for a fixed `n`, as we scan along `hist`, if there is a drop in `hist[m]`, such that `hist[n] > hist[m]` then :
- no need to consider `rect(n,m)`, `rect(n,m+1)`, `rect(n,m+2)` and so on
- only need to consider `rect(n,n)`, `rect(n,n+1)`, …, `rect(n,m-1)`
- hence we construct `stack` to store an increasing profile along the linear scan

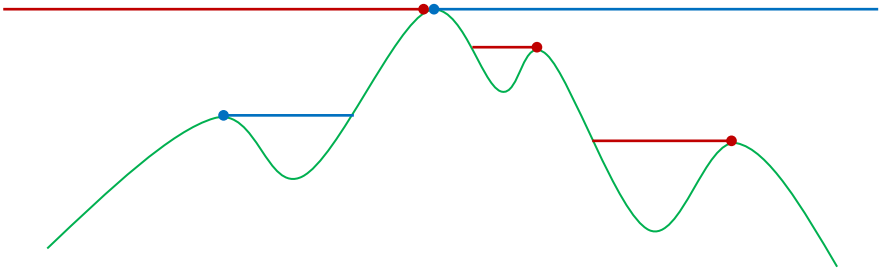
```
def biggest_rectangle(hist) :
    max_rect_area  = 0
    max_rect_start = 0
    stack = [[0, hist[0]]]
    for n in range(1,len(hist)) :
        m =-1

        # [Part1] area from m to n-1, where n<m (it is a fault if you consider area from m to n)
        while len(stack) > 0 and hist[n] < stack[-1][1] :
            m,hm = stack.pop(-1)
            area = hm * (n-m) # BUG : use "hm" instead of "hist[m]", as it changes across iteration
            if area > max_rect_area :
                max_rect_area = area
                max_rect_start = m

        # BUG : Don't miss this!!!
        if m>-1 : stack.append([m, hist[n]])
        else    : stack.append([n, hist[n]])

    # [Part2] pop until empty
    while len(stack) > 0 :
        m,hm = stack.pop(-1)
        area = hm * (len(hist)-m)
        if area > max_rect_area :
            max_rect_area = area
            max_rect_start = m
    return (max_rect_start, max_rect_area)
```

Please watch https://www.youtube.com/watch?v=zx5Sw9130L0

Q8.4 Trapping water problem

Unlike biggest rectangle, the trapping water problem is easier, the solution is to consider it as a landscape :

- place a spot light on the LHS of the array, cast a *shadow to RHS*
- place a spot light on the RHS of the array, cast a *shadow to LHS*
- hence it is two scans, once from LHS, once from RHS
- for each bucket, find the minimum of two shadows, *water height = min(blue,red) - landprofile*



In practice, how can we trace the shadow? **No stack technique is needed**. Just running maximum is fine.

Q9.1 Order statistic in *O(N)*

Given a unsorted vector of integers, find the smallest and second smallest integer in *O(N)*.

```
def order_stat(vec) :
    min0 = min(vec[0], vec[1])
    min1 = max(vec[0], vec[1])
    for n in range(2,len(vec)) :
        if vec[n] < min0 :
            min1 = min0
            min0 = vec[n]
        elif vec[n] < min1 :
            min1 = vec[n]
    return [min0, min1]
```

Given a unsorted vector of integers, find the K*th* smallest integer in *O(N)*.

- By generalizing the idea in previous part to the K*th* smallest integer, we will end up in $O(N^2)$.
- By sorting the vector and pick the K*th* integer from sorted vector, we will end up in *O(NlogN)*.
- However we can further optimize by consider the concept of various sorting :

→ after *nth* iteration of `select sort` : *1st – nth* optimal items are already in the front of container (this is too much)

→ after *nth* iteration of `bubble sort` : *1st – nth* optimal items are already in the front of container (this is too much)

→ after *nth* iteration of `insert sort` : *1st n* elements in container are sorted (but they are not in ground true position)

→ after *nth* iteration of `quick sort` : *n* items lying correct in their places (that's what we need, K*th* item in its place)

```
def order_stat(vec, k) :
    front = 0
    back = len(vec)-1
    while front != back :
        i = front
        j = back
        while i!=j :
            if vec[i] <= vec[j] : j = j-1
            else :
                temp = vec[i] // This loop works even for i+1=j.
                vec[i] = vec[j]
                vec[j] = vec[i+1]
                vec[i+1] = temp
                i = i+1

        # instead of calling recursively ...
        # quick(vec[:i]) & quick(vec[i+1:])

        # Here is a slight modification from bisection in algo.doc. Please note i==j.
        if k == i : return vec[i]
        if k < i :
            if front+1==back and i==front : back = i   // avoid access out of boundary
            else                          : back = i-1 // more aggressive update
        else
            if front+1==back and i==back : front = i   // avoid access out of boundary
            else                         : front = i+1 // more aggressive update
    return vec[front]
```

Why is this algorithm *O(N)*?

$$N + N/2 + N/4 + N/8 + \ldots = 2N$$

Comparing to *O(NlogN)* in `quick sort`

$$N + (N/2)\times 2 + (N/4)\times 4 + \ldots = \underbrace{N + N + N + \ldots}_{\log_2 N}$$

Q9.2 Minimum number of adjacent swaps given unsorted vector with limited shuffle

Given a sorted vector, it is shuffled through a series of adjacent swaps, each item can be swapped forward at most twice, but can be swapped backward for unlimited number of times. What is the minimum of adjacent swaps needed to sort the shuffled vector?

```
def minimum_adjacemt_swaps(q) : # q[n] is original position, n is new position, where n >= q[n]-2 for all n (max 2 bribes)
    num_bribes = 0
    for n in range(len(q)) :
        if n < q[n]-2 :
            print('violate bribing twice at max assumption')
            return None

    // *** Slow method *** //
    for n in range(len(q)) :
        for m in range(0, n) :
            if q[m] > q[n] : ++num_bribes

    // *** Fast method *** //
    for n in range(len(q)) :
        for m in range(max(0, q[n]-2), n) :
            if q[m] > q[n] : ++num_bribes

    return num_bribes
```

```
// *** Slow method *** //
for n in range(len(q)) :
    for m in range(n+1, len(q)) :
        if q[m] < q[n] : ++num_bribes

// *** Slow method *** //
for n in range(len(q)) :
    for m in range(n+1, lenq(q)) :
        if q[m] < q[n] : ++num_bribes
```

We can do the optimization here, because the IF condition in the following loop is never true. Otherwise

```
for m in range(0, max(0, q[n]-2)) :
    if q[m] > q[n] : ++num_bribes
```

```
if it is true, then q[m]-2 > q[n]-2 > m
which violates m > q[m]-2 assumption
```

Q10.1 Set of subsets

Given a string of `a` and `b`, find all non-contiguous substrings. For example, given `aaba`, we have `aaba`, `aab`, `aba`, `ab`, `ba`, `aaa`, `aa`, `a`.

```
def find_substr(str, substrs) :
    if len(str) == 0 : return

    temp = set()
    find_substr(str[1:], temp)

    for s in temp :
        substrs.add(s)
        substrs.add(str[0]+s)
    substrs.add(str[0:1]) # Dont forget this
```

Q10.2 Layer average in a binary search tree of integers

```
template<typename T> void find_layer_average(const node<T>* root, std::vector<std::pair<int, int>>& sum_and_count)
{
     std::queue<std::pair<const node<int>*, int>> q; // queue for region growing (tree traversal), include node* and layer-info
     if (root) q.push(std::make_pair(root, 0)); else return;

     while (!q.empty()) // This is iterative tree-traversal (not recursive tree-traversal)
     {
          auto [this_node, layer] = q.front(); q.pop();
          while (layer >= sum_and_count.size()) sum_and_count.push_back(std::make_pair(0, 0));
          sum_and_count[layer].first  += this_node->value;
          sum_and_count[layer].second += 1;                                    ⇐ core implementation
          if (this_node->lhs) q.push(std::make_pair(this_node->lhs, layer+1));
          if (this_node->rhs) q.push(std::make_pair(this_node->rhs, layer+1));
     }
}
```

Q10.3 String matching algorithm

We have greedy algorithm in *O(N)*, yet it fails when template has repeated pattern like `ABCABXYZ`. Then we have *O(N×K)* conservative algorithm, which is always correct but slow. Finally, we have *O(N)* KMP algorithm which makes use of backtrace. The 3 algorithms differ by the updating equations for `m` and `n` only.

```
def string_match(s, t, algo) :
    if (algo == KMP) btrace = creat_backtrace(t)
    result = []
    n = 0 # index in string
    m = 0 # index in template
    while n != len(s) :
        if s[n] == t[m] and m < len(t)-1 :
            n = n+1
            m = m+1
        elif s[n] == t[m] and m == len(t)-1 :
            result.append(n-m)
            # greedy implement      # conservative       # KMP algo
            n = n+1                 n = n-m+1            n = n+1
            m = 0                   m = 0                m = btrace[m]
        elif m > 0 :
            # greedy implement      # conservative       # KMP algo
            n = n+1                 n = n-m+1            n = n+1
            m = 0                   m = 0                m = btrace[m-1]
        else :
            n = n+1
            m = 0
    return result

def creat_backtrace(t) :                                 # btrace[m] = max num of self-matched char if matching ends at m
    btrace = [0] * len(t)                                # size(btrace) = size(template), most entries in btrace are 0
                                                         # For example, template ABCABXY
    for shift in range(1,len(t)) :                       #              btrace: {0001200}
        for m in range(len(t)-shift) :                   # For example, template ABCABC
            if t[m] != t[shift+m] : break                #              btrace: {000123}
        else : m = len(t)-shift
        btrace[shift+m-1] = max(btrace[shift+m-1], m)
    return btrace
```