Advanced algorithms

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

Linked lists

1.1 Floyd's cycle finding algorithm (Hare-Tortoise algorithm)

Floyd's cycle finding algorithm or Hare-Tortoise algorithm is a pointer algorithm that uses only two pointers, moving through the sequence at different speeds. This algorithm is used to find a loop in a linked list. It uses two pointers one moving twice as fast as the other one. The faster one is called the fast pointer and the other one is called the slow pointer.

While traversing the linked list one of these things will occur-

- The Fast pointer may reach the end (NULL) which shows that there is no loop in the linked list.
- The Fast pointer again catches the slow pointer at some time therefore a loop exists in the linked list.

```
0  bool hasCycle(ListNode *head) {
1      if (!head) return false;
2      ListNode* slow = head, *fast = head;
3
4      while (fast && slow && slow->next && fast->next &&
   ↪   fast->next->next) {
5          slow = slow->next;
6          fast = fast->next->next;
7          if (slow == fast) return true;
8      }
9      return false;
10 }
```

1.1.1 Location of where the cycle begins

Floyd's algorithm (also known as the "tortoise and hare" algorithm) not only detects a cycle in a linked list but can also identify the exact node where the cycle begins.

If there is a cycle, the fast pointer will eventually "lap" the slow pointer, meaning they will meet at some node inside the cycle. If no cycle exists, the fast pointer will reach the end of the list (i.e., a null reference).

Once a meeting point is found, reset the slow pointer to the head of the list while keeping the fast pointer at the meeting point. Now move both pointers one step at a time.

The point at which the two pointers meet again is the start of the cycle.

```cpp
ListNode* floyd(ListNode* head) {
    ListNode* slow = head, *fast = head;

    while (fast) {
        slow = slow->next;

        if (fast->next && fast->next->next) {
            fast=fast->next->next;
            if (slow == fast) break;
        }
        else fast=nullptr;

    }
    if (!fast) return nullptr;

    slow = head;
    while (slow != fast) {
        slow=slow->next;
        fast=fast->next;
    }

    return slow;
}

int main(int argc, const char** argv) {

    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    ListNode* cycle_begin = head->next;
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = cycle_begin;

    ListNode* f = floyd(head);
    if (f) cout << f->val;
    else cout << "No cycle";
}
```

1.1.2 Proof of Floyd's algorithm

Let

$$\begin{aligned} L &= \text{ Distance from head to the beginning of the cycle} \\ C &= \text{ Length of the cycle} \\ K &= \text{ Distance from start of cycle to meeting point.} \end{aligned}$$

Once the slow pointers meet, the slow pointer has moved $L + K$ times, whereas the fast pointer has moved $L + K + nC$, where n is the number of completed cycles. Since the fast pointer moves twice as fast, we have

$$2(L + K) = L + K + nC.$$

Solving for L, we get

$$L = nC - K.$$

Reset the slow pointer to the head of the list, move both pointers at the same rate (rate of the slow pointer) L times. The slow pointer will be at the start of the cycle, the fast pointer (which started K nodes into the cycle) will have moved

$$(L + K) \mod C.$$

But, $L = nC - K$. Thus,

$$(L + K) \mod C = (nC - K + K) \mod C = nC \mod C = 0.$$

Thus, the fast pointer will also be at the start of the cycle, since it is zero nodes into the cycle. ■

1.1.3 Practical example, find duplicate

Given an array of integers nums containing $n + 1$ integers where each integer is in the range $[1, n]$ inclusive.

There is only one repeated number in nums, return this repeated number.

You must solve the problem without modifying the array nums and using only constant extra space.

```
0  int findDuplicate(vector<int>& nums) {
1      int slow = 0, fast = 0;
2      do {
3          slow = nums[slow];
4          fast = nums[nums[fast]];
5      } while (slow != fast);
6
7      slow = 0;
8      do {
9          slow = nums[slow];
10         fast = nums[fast];
11     } while (slow != fast);
12     return slow;
13 }
```

Range queries

In a range query, our task is to calculate a value based on a subarray of an array. Typical range queries are

- $sumq(a, b)$: calculate the sum of values in range $[a, b]$
- $minq(a, b)$: find the minimum value in range $[a, b]$
- $maxq(a, b)$: find the maximum value in range $[a, b]$

A simple way to process range queries is to use a loop that goes through all array values in the range.

This works in $O(n)$ time, where n is the size of the array. Thus, we can process q queries in $O(nq)$ time using the function. However, if both n and q are large, this approach is slow. Fortunately, it turns out that there are ways to process range queries much more efficiently.

2.1 Static array queries

We first focus on a situation where the array is static, i.e., the array values are never updated between the queries. In this case, it suffices to construct a static data structure that tells us the answer for any possible query.

2.1.1 Sum queries

We can easily process sum queries on a static array by constructing a prefix sum array. Each value in the prefix sum array equals the sum of values in the original array up to that position. That is, the value at position k is $\text{sum}_q(0, k)$

we can calculate any value of $\text{sum}_q(a, b)$ in $O(1)$ time as follows:

$$\text{sum}_q(a, b) = \text{sum}_q(0, b) - \text{sum}_q(0, a - 1).$$

We define $\text{sum}_q(0, -1) = 0$, so that the formula holds when $a = 0$

2.1.2 Minimum and maximum queries: Sparse tables

Minimum queries are more difficult to process than sum queries. Still, there is a quite simple $O(n \lg n)$ time preprocessing method after which we can answer any minimum query in $O(1)$ time

Note that since minimum and maximum queries can be processed similarly, we can focus on minimum queries.

The idea is to precalculate all values of $\min_q(a, b)$ where $b - a + 1$ (the length of the range) is a power of two. For example, for the array

$$\text{A} := [1,3,4,8,6,1,4,2]$$

The following values are calculated

a	b	$\min_q(a,b)$
0	0	1
1	1	3
2	2	4
3	3	8
4	4	6
5	5	1
6	6	4
7	7	2

a	b	$\min_q(a,b)$
0	1	1
1	2	3
2	3	4
3	4	6
4	5	1
5	6	1
6	7	2

a	b	$\min_q(a,b)$
0	3	1
1	4	3
2	5	1
3	6	1
4	7	1
0	7	1

.

The number of precalculated values is $O(n \lg n)$, because there are $O(\lg n)$ range lengths that are powers of two. The values can be calculated efficiently using the recursive formula

$$\min_q(a,b) = \min(\min_q(a, a+w-1), \min_q(a+w, b))$$

where $b-a+1$ is a power of two and $w = \frac{b-a+1}{2}$, calculating all those values takes $O(n \lg n)$ time.

After this, any value of $\min_q(a,b)$ can be calculated in $O(1)$ time as a minimum of two precalculated values. Let k be the largest power of two that does not exceed $b-a+1$. We can calculate the value of minq(a,b) using the formula

$$\min_q(a,b) = \min(\min_q 9a, a+k-1, \min_q(b-k+1, b)).$$

In the above formula, the range $[a,b]$ is represented as the union of the ranges $[a, a+k-1]$ and $[b-k+1, b]$, both of length k

Note that we have the min for both these ranges, since

$$a+k-1-a+1 = k,$$
$$b-(b-k+1)+1 = k$$

and k is a power of two. Also, we have that

$$k \leqslant L$$

where $L = (b-a+1)$, it follows then that

$$L < 2k.$$

So,

$$k \leqslant L < 2k.$$

Thus, for our split $A[a, a+k-1]$ and $A[b-k+1, b]$, both segments have length k, so the sum of the number of elements in both segments is $2k$. If segment one starts at a, and segment two ends at b, and the total is $2k$, then the union must capture the original subarray $A[a,b]$, since $L < 2k$.

The structure that we store the preprocessed work in is called a **sparse table**.

We create a matrix table $st[i][j]$ such that

- i represents the starting index in the array
- j represents the interval length as a power of 2. That is, the interval covers 2^j elements

The idea is that $st[i][j]$ stores the minimum value in the subarray from index i to index $i + 2^j - 1$. If the ending index is k, then

$$2^j = k - i + 1 \implies k = 2^j + i - 1 = i + 2^j - 1.$$

For every i in the range 0 to $n - 1$,

$$st[i][0] = A[i].$$

This corresponds to an interval of length 1 (2^0).

For every j from 1 up to

```cpp
void buildSparseTable(const vector<int>& v, vector<vector<int>>&
    st, vector<int>& log) {
    int n = v.size();
    int K = floor(log2(n)) + 1;

    // Resize and initialize the sparse table.
    st.assign(n, vector<int>(K));

    // Compute log values for each i.
    log.resize(n + 1);
    log[1] = 0;
    for (int i = 2; i <= n; i++) {
        log[i] = log[i / 2] + 1;
    }

    // Initialize st[i][0] = v[i].
    for (int i = 0; i < n; i++) {
        st[i][0] = v[i];
    }

    // Build sparse table: combine intervals.
    for (int j = 1; j < K; j++) {
        for (int i = 0; i + (1 << j) <= n; i++) {
            st[i][j] = std::min(st[i][j - 1], st[i + (1 << (j -
    1))][j - 1]);
        }
    }
}

int query(int L, int R, const vector<vector<int>>& st, const
    vector<int>& log) {
    int len = R - L + 1;
    int j = log[len];
    return std::min(st[L][j], st[R - (1 << j) + 1][j]);
}
```

2.2 Difference arrays

Difference arrays are a useful technique in data structures and algorithms that allow for efficient range updates on an array.

A difference array is derived from an original array. It stores the difference between consecutive elements. For an array A of size n, the difference array D is defined as:

$$\begin{aligned} D[0] &= A[0] \\ D[i] &= A[i] - A[i-1] \quad \text{for} i \in \{1, 2, ...n-1\} \end{aligned}$$

The key idea is that if you want to add a constant value x to a range from index l to r in A, you can update the difference array D as follows

$$\begin{aligned} D[l] &+= x \\ D[r+1] &-= x \quad \text{if } r+1 \text{ in bounds} \end{aligned}$$

After processing all such range updates, you can reconstruct the final array by computing the prefix sum of D

$$\begin{aligned} A[0] &= D[0] \\ A[i] &= A[i-1] + D[i] \quad \text{for } i \in \{1, 2, ..., n-1\} \end{aligned}$$

Range updates that would normally take $O(n)$ time each can be done in $O(1)$ time

Consider an example

$$A = [1, 2, 3, 4, 5]$$

Thus,

$$D = [1, 1, 1, 1, 1]$$

With query $Q(1, 3, 3)$, of form $Q(l_i, r_j, v)$. Thus, $Q(1, 3, 3) = A[i] += 3$ for $i \in \{1, 2, 3\}$

Applying $D[l] = D[1] += 3$, $D[r+1] = D[4] -= 3$, we get

$$D = [1, 4, 1, 1, -2]$$

Then we reconstruct A using prefix sums $A[i] = A[i-1] + D[i]$, and we get

$$A = [1, 5, 6, 7, 5]$$

if r is the last index in the original array, then $r+1$ will be out of bounds. In that case, you simply omit the subtraction step because there's no element beyond the last index. This naturally means that the update applies to all elements from l to the end of the array. Consider the query $Q(1, 4, 3)$ on the array $A = [1, 2, 3, 4, 5]$. The difference array is

$$D = [1, 4, 1, 1, 1]$$

Notice that since $D[r+1] = D[5]$ is out of bounds, we only need to do $D[l] = D[1] += 3$. The final array is

$$A[1, 5, 6, 7, 8]$$

2.2.1 Why it works

Let's first consider the property between the difference array D and the array A

$$\begin{aligned} D[0] &= A[0] \\ D[i] &= A[i] - A[i-1] \end{aligned}$$

Thus,

$$A[i] = D[i] + A[i-1]$$

Suppose we have the array $A[a, b, c, d, e]$, and we want to add k to $A[i]$, for $i \in \{1, 2, 3\}$, we have

$$A[a, b+k, c+k, d+k, e]$$

Then,

$$\begin{aligned} &D[a, b+k-a, b+k-c-k, d+k-c-k, e-d-k] \\ &= [a, b-a+k, b-c, d-c, e-k] \end{aligned}$$

We see the only changes to D are $D[l]$, and $D[r+1]$

$$1 + 2 + 3 + \cdots$$

Intervals

Interval problems are a class of problems in data structures and algorithms that deal with ranges or segments, typically represented by start and end points. These problems often require determining relationships between intervals—such as overlapping, containment, or disjointness—and finding optimal ways to select, merge, or query these intervals.

Common problem types are

- **Interval Scheduling:** Select the maximum number of non-overlapping intervals from a given set. A classic example is scheduling meetings in a conference room.
- **Interval Partitioning:** Assign intervals to different resources (like classrooms or machines) such that no overlapping intervals share the same resource, while minimizing the total number of resources used.
- **Merging Intervals:** Combine overlapping intervals into a single continuous interval. This is often required in scenarios like consolidating time ranges or segments.
- **Interval Intersection/Union:** Find the common overlapping parts (intersection) of intervals or the total span covered by intervals (union).

Data structures and techniques that are relevant are

- **Greedy Algorithms:** For many interval scheduling problems, a greedy approach (such as selecting the interval that ends earliest) leads to an optimal solution.
- **Sorting:** Many interval problems begin with sorting the intervals based on start or end times to simplify subsequent processing.
- **Interval Trees:** These are balanced trees designed to hold intervals and allow efficient querying, such as finding all intervals that overlap with a given query interval.
- **Segment Trees:** Useful for range queries and updates, segment trees can efficiently manage intervals when dealing with aggregated data like sums or minima/maxima over a range.

3.1 Merge intervals

The merge intervals problem is a problem that takes overlapping intervals, and merges the ones that overlap. For example,

$$[1,3],[2,6],[8,10],[15,18] \to [1,6],[8,10],[15,18]$$

We note that intervals with common endpoints are considered to be overlapping. For example, $[1,4],[4,5]$ would be merged into $[1,5]$

```
    class Solution {
        public:
        vector<vector<int>> merge(vector<vector<int>>&
↪   intervals) {
            std::sort(intervals.begin(), intervals.end());

            vector<vector<int>> m;
            int curr_start = intervals[0][0];
            int curr_end = intervals[0][1];

            for (int i=1; i<(int)intervals.size(); ++i) {
                int next_left = intervals[i][0], next_right =
↪   intervals[i][1];

            if (next_left >= curr_start && next_left <=
↪   curr_end) {
                curr_end = std::max(curr_end, next_right);
            } else {
                m.push_back({curr_start,curr_end});
                curr_start = next_left, curr_end = next_right;
            }
        }
        m.push_back({curr_start, curr_end});
        return m;
    }
};

// [[1,3],[2,6],[8,10],[15,18]] -> [[1,6],[8,10],[15,18]]
```

Time complexity is $O(n \lg n)$, space complexity is $O(n)$

3.2 Line sweep (sweep line)

Line Sweep (or Sweep Line) is an algorithmic technique where we sweep an imaginary line (x or y axis) and solve various problem.

3.2.1 Merge intervals with line sweep

```cpp
class Solution {
    public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        map<int, int> line;
        for (const auto& e : intervals) {
            ++line[e[0]];
            --line[e[1]];
        }

        int count=0, start=0;
        vector<vector<int>> res;
        for (auto& [key, value] : line) {
            if (count==0) {
                start = value;
            }
            count+=value;

            if (count == 0) {
                res.push_back({start, value});
            }
        }
        return res;
    }
};
```

For every interval [start, end] in the input:

> ++line[e[0]]; increments the count at the start point. –line[e[1]]; decrements the count at the end point.

This records ”events” along the number line: a positive event when an interval begins and a negative event when an interval ends. The map keeps these events in sorted order by key (the coordinate).

- **count:** Tracks how many intervals are currently active as we sweep through the line.
- **ans:** Will store the resulting merged intervals.
- **start:** Used to mark the beginning of a new merged interval.

The for loop goes through each key-value pair in the map line in ascending order of the coordinate (since map is sorted by keys).

When count is 0 before processing an event, this indicates that no interval is active at that moment. The code then sets start to the current coordinate (i.first), which will be the beginning of a new merged interval.

count += i.second; adjusts the active interval count based on the event at that coordinate.

A positive count means one or more intervals are active, while a count returning to 0 means all intervals that started have now ended.

When count becomes 0, it indicates that a merged interval has ended at the current coordinate.

The code then pushes the interval [start, i.first] into ans.

Bitwise operations (and tricks)

4.1 Bitwise power of two

To check if a positive integer n is a power of two. That is, $n = 2^x$ for some $x \in \mathbb{N}$, we check

```
if (n & n-1 == 0) {
    // n is a power of two
}
```

If n is a power of two, it can be written as 2^k . Its binary form will have a single '1' followed by k zeros

When you subtract 1 from a power of two, the binary representation becomes a sequence of k ones.

Consider $1000 - 0001$. We can convert 0001 to its twos complement representation, and change the subtraction to addition. 0001 in its two complement representation is 1110. Thus, we have

$$\begin{array}{cccccc} & 1 & 0 & 0 & 0 & \\ - & 0 & 0 & 0 & 1 & \\ \hline & & & & & \end{array} = \begin{array}{cccccc} & & 1 & 0 & & \\ & 1 & 0 & 0 & 0 & \\ + & 1 & 1 & 1 & 1 & \\ \hline & 0 & 1 & 1 & 1 & \end{array}$$

The operation $n\&(n-1)$ compares each bit of n and $n-1$. Since n has a single '1' in a position where $n-1$ has a '0' (and vice versa for all lower bits), none of the corresponding bits are '1' at the same time. Therefore, the result is 0.

If n is not a power of two, its binary representation will have more than one '1'. Subtracting 1 will not clear all the '1' bits when you perform the AND operation, so the result will be non-zero.

If $n = 0$, $n\&(n-1)$ will be zero, but 0 is not a power of two, so we require $n > 0$.

4.2 Bitwise parity check

We can check the parity of a number by observing if the rightmost bit is set. If the rightmost bit is one, then the number is odd, if its zero, then the number is even.

```
bool parity(int n) {
    // Equivalent result: return n % 2
    return n & 1;
}
```

As an example, consider 4 & 1... That is, 0100 & 0001, we see

$$\begin{array}{cccccc} & 0 & 1 & 0 & 0 \\ & 0 & 0 & 0 & 1 \\ \hline \& & 0 & 0 & 0 & 0 \end{array}.$$

And for 4 & 1 = 0101 & 0001, we have

$$\begin{array}{cccccc} & 0 & 1 & 0 & 1 \\ & 0 & 0 & 0 & 1 \\ \hline \& & 0 & 0 & 0 & 1 \end{array}.$$

4.3 Bitwise divide and multiply by two

For this we can use left and right shifts (by one). To multiply a number by 2, we shift left by one. For a divide by two, we shift right by one

```
0  int x = 50
1  cout << (x << 1) << endl; // 100
2  cout << (x >> 1) << endl; // 25
```

Math related things

5.1 Floors and Ceils

5.1.1 Defining the notation

We have

$$\lfloor x \rfloor = \text{Max}\{n : \ n \in \mathbb{Z}, \ n \leqslant x\}$$
$$\lceil x \rceil = \text{Min}\{n : \ n \in \mathbb{Z}, \ n \geqslant x\}.$$

5.1.2 Equating ceils and floors

We two integers a, b, we have the property

$$\left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a+b-1}{b} \right\rfloor.$$

We have that

$$a = bq + r \quad 0 \leqslant r < b.$$

where q is the integer quotient $q = \frac{a}{b}$ and r is the remainder

The ceil $\left\lceil \frac{a}{b} \right\rceil$ is therefore

$$\left\lceil \frac{a}{b} \right\rceil = \begin{cases} q & \text{if } r = 0 \\ q+1 & \text{if } r > 0 \end{cases}.$$

To ensure that we round up when there is a nonzero remainder, we add $b-1$ to a, this yields

$$a + b - 1 = bq + r + b - 1 = b(q+1) + (r-1).$$

So, if $r = 0$, we have that

$$\begin{aligned} \lfloor a+b-1 \rfloor &= \lfloor b(q+1) - 1 \rfloor \\ \implies \left\lfloor \frac{a+b-1}{b} \right\rfloor &= \left\lfloor q + 1 - \frac{1}{b} \right\rfloor \\ &= \left\lfloor q + \frac{b-1}{b} \right\rfloor. \end{aligned}$$

Since $\frac{b-1}{b} < 1$, so

$$\left\lfloor \frac{a+b-1}{b} \right\rfloor = q.$$

When $r > 0$, we have

$$\lfloor a+b-1 \rfloor = \lfloor b(q+1)+(r-1) \rfloor$$
$$\implies \left\lfloor \frac{a+b-1}{b} \right\rfloor = \left\lfloor q+1+\frac{r-1}{b} \right\rfloor.$$

and since $0 < r < b$, $\frac{r-1}{b} < 1$, so

$$\left\lfloor \frac{a+b-1}{b} \right\rfloor = q+1.$$

So, we can use this property in algorithms to replace a call to a ceil function, and instead use integer division. Consider the following example, suppose we need to compute

$$\left\lceil \frac{n}{2} \right\rceil.$$

Then, we have

$$\left\lceil \frac{n}{2} \right\rceil = \left\lfloor \frac{n+2-1}{2} \right\rfloor = \left\lfloor \frac{n+1}{2} \right\rfloor.$$

```
int n = 15
cout << ceil((double)n/2) << endl;
cout << (n+1)/2 << endl; // Equivalent
```

5.2 Another useful property

For an integer n, we have that

$$\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = n.$$

If n even, then $n = 2k$ for some integer k, then

$$\begin{aligned}\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil &= \left\lfloor \frac{2k}{2} \right\rfloor + \left\lceil \frac{2k}{2} \right\rceil \\ &= \lfloor k \rfloor + \lceil k \rceil = 2k = n.\end{aligned}$$

If n is odd, then $n = 2k+1$ for some integer k, and

$$\begin{aligned}\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil &= \left\lfloor \frac{2k+1}{2} \right\rfloor + \left\lceil \frac{2k+1}{2} \right\rceil \\ &= \left\lfloor k+\frac{1}{2} \right\rfloor + \left\lceil k+\frac{1}{2} \right\rceil = k+(k+1) = 2k+1 = n.\end{aligned}$$

Thus,

$$\left\lceil \frac{n}{2} \right\rceil = n - \left\lfloor \frac{n}{2} \right\rfloor.$$

```
int n = 25;
cout << (ceil((double)n/2)) << endl;
cout << (n - n/2) << endl; // Equivalent
```

5.3 Counting

5.3.1 Counting number of subarrays

Let $A = [a_1, a_2, a_3, ..., a_n]$, suppose we want to count the total number of possible contiguous subarrays. For example, if $A = [1, 2, 3, 4]$, then define ω to be the set of all subarrays. We have

$$\omega = \{[1], [2], [3], [4], [1, 2], [2, 3], [3, 4], [1, 2, 3], [2, 3, 4], [1, 2, 3, 4]\}.$$

Thus, $|\omega| = 10$. In general, we have that

$$|\omega| = \sum_{i=1}^{n}(n - i + 1) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

If we include $\varnothing$ to be a valid subarray, then $|\omega| = \frac{n(n+1)}{2} + 1$

For each subarray size i, starting from 1 to n, we consider the number of starting points, but we must be able to include i elements. Thus, the last element that can be a starting point for a subarray of size i is at index $n - i + 1$. Thus, there are $n - i + 1$ valid subarrays of size i

5.3.2 Counting number of subarrays with length a power of two

Consider an array $A = [a_1, a_2, a_3, ..., a_n]$, let $\phi[i, j]$ be a subarray of A with $0 \leqslant i \leqslant j \leqslant n$, suppose we want to count the number of subarrays such that the length $j - i + 1$ is a power of two.

So, for $L = j - i + 1$, we require that $L = 2^\ell$, for $\ell \in \mathbb{Z}^+$. We have that 2^ℓ must satisfy

$$2^\ell \leqslant n \implies \ell \leqslant \lg n.$$

So, by the logic in the previous subsection, we have that

$$|\omega| = \sum_{\ell=0}^{\lfloor \lg n \rfloor} n - 2^\ell + 1 = \Theta(n \lg n).$$

And, the number of subarrays with length a power of two is $O(\lg n)$, or more precisely $\Theta(\lg n)$.

Math algorithms

6.1 Euclidean GCD Algorithm

The GCD of two integers a and b (with $a \leqslant b$) is the largest integer that divides both a and b. The Euclidean algorithm is based on the principle that

$$\gcd(a, b) = \gcd(b, a \bmod b).$$

This means that the GCD of two numbers doesn't change if the larger number is replaced by its remainder when divided by the smaller number. You keep repeating this until the remainder is 0, and the GCD will be the last non-zero remainder

```
int gcd(int a, int b)  {
    if (!b) return a;

    return gcd(b, a%b);
}
// An iterative approach
int gcd(int a, int b ) {
    if (b < a) {
        b = std::exchange(a,b);
    }
    while (b != 0) {
        a = std::exchange(b, a % b);
    }
    return a;
}
```

6.1.1 GCD of some integer collection

We can apply the above algorithm to find the GCD of a collection of elements

```
int gcd_set(const vector<int>& v) {
    if (v.empty()) return 0;
    int result = v[0];
    for (int i=1; i<(int)v.size(); ++i) {
        result = gcd(result, v[i]);
        // Return early
        if (result == 1) return result;
    }
    return result;
}
```

6.2 Fibonacci numbers in constant time

The formula for the nth Fibonacci number $F(n)$ can be derived using Binet's formula, which expresses the Fibonacci sequence in terms of powers of the golden ratio.

$$F(n) = \frac{\phi^n - \psi^n}{\sqrt{5}}.$$

Where ϕ is the golden ration $\frac{1+\sqrt{5}}{2} \approx 1.618$, and ψ is the conjugate of the golden ration $\psi = \frac{1-\sqrt{5}}{2} \approx -0.618$

6.3 Sterlings factorial approximation

Stirling's Approximation provides an approximation for factorials, particularly useful for large values of n. The formula is:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

6.4 square roots

Given a non-negative integer x, return the square root of x rounded down to the nearest integer. The returned integer should be non-negative as well.

Note: You must not use any built-in exponent function or operator.

6.4.1 Naive $O(\sqrt{n})$

```
0  typedef long long dword;
1  class Solution {
2      public:
3      int mySqrt(int x) {
4          dword last{};
5          for (dword i=0; i<=x; ++i) {
6              if (i*i == x) return i;
7              else if (i*i > x) return last;
8              last = i;
9          }
10         return 0;
11     }
12 };
```

For each number $i \in [0, x]$, we simply check either the square of i is precisely x, or $i^2 > x$, in which case we return the value of i before the current iteration.

This algorithm runs in $O(\sqrt{n})$ time because the for loop runs at most $\sqrt{x}$ times.

Note that we start i at one and go up to including x specifically to handle the case where $x = 1$. If $x = 0$, the for loop never runs and the default initialized dword gets returned (which is correct)

6.4.2 $O(\lg(n))$ binary search approach

```
typedef long long dword;
class Solution {
    public:
    int mySqrt(int x) {
        dword start{}, end = x, mid, res;
        while (start <= end) {
            mid = (start + end) / 2;

            if (mid * mid > x) {
                end = mid-1;
            } else if (mid * mid < x) {
                start = mid + 1;
                res = mid;
            } else {
                return mid;
            }
        }
        return res;
    }
};
```

6.5 Fast exponentiation ($O(\lg(n))$)

To perform exponentiation faster than $O(n)$, use Exponentiation by Squaring, which runs in $O(logn)$ time.

This method efficiently computes a^b using the following rules:

1. If b is even

$$a^b = \left(a^{\frac{b}{2}}\right)^2 = \left(a^2\right)^{\frac{b}{2}}$$

2. If b is odd

$$a^b = a \cdot a^{b-1}$$

```
0       long long fastExponentation(long long a, long long b) {
1           long long result = 1;
2           while (b > 0) {
3               if (b % 2 == 1) {
4               result *= a;
5           }
6           a*=a;
7           b/=2;
8       }
9       return result;
10  }
```

6.5.1 Recursive

```
    class Solution {
        public:
        double r_fe(double& x, long long& n, double& res) {
            if (n <= 0) return res;

            if (n%2 == 1) {
            res*=x;
            --n;
        }
        n/=2;
        x*=x;

        return r_fe(x,n,res);
    }

    double myPow(double x, long long n) {
        double res{1.0};
        if (n < 0) {
            n = -n;
            return 1 / r_fe(x,n,res);
        }
        return r_fe(x,n,res);
    }
};
```

6.6 Fast modular exponentiation

Fast modular exponentiation, also known as binary exponentiation or exponentiation by squaring, is an algorithm used to efficiently compute $a^b \mod m$ without directly calculating a^b (which can be very large)

It works similar to the *fast exponentiation* algorithm above, but takes mods along the way.

```
0  #define ll long long
1  ll modExp(ll a, ll b, ll mod) {
2      ll result = 1;
3      a %= mod;
4      while (b > 0) {
5          if (b & 1)
6          result = (result * a) % mod;
7          a = (a * a) % mod;
8          b >>= 1;
9      }
10     return result;
11 }
```

Number Theory

7.1 Getting prime factors of a number

```
    vector<int> prime_factors(int n) {
        vector<int> factors;
        for (int i=2; i<=sqrt(n); ++i) {
            if (n % i == 0) {
            while (n%i == 0) {
            factors.push_back(i);
            n/=i;
        }
    }
}
// If n was originally prime, it has no prime factors other than
↪  1 and itself.
if (n >= 2) factors.push_back(n);
return factors;
}


int main(int argc, const char** argv) {
    vector<int> res = prime_factors(16);
    cout << res[0];
    for (int i=1; i<(int)res.size();++i) cout << "*" << res[i];

    return EXIT_SUCCESS;
}
```

The algorithm works by testing every possible factor starting at 2 and moving upward. When it finds a number i that divides n, it repeatedly divides n by i until i no longer divides it. Because it starts with the smallest numbers (which are prime), any composite number would already have been broken down into its prime components. In other words, if a composite factor divides n, one of its smaller prime factors would have divided n first and removed that component. Thus, only prime factors remain to be printed.