NOI.PH Training: Week 4

Jared Guissmo Asuncion

Contents

Chapter 1

Divide and Conquer

The divide-and-conquer paradigm is a useful technique in solving certain problems. The idea of this paradigm is simple. Suppose we have a problem whose input size is N. Then, we must

- find a way to split the big problem of size N into two smaller subproblems of size $N/2$, and
- find a way to obtain a solution for the big problem using the solutions of two smaller subproblems.

1.1 Modular Exponentiation

A good first example is modular exponentiation, but first we need some facts about integer division.

Theorem 1.1 (division algorithm in $\mathbb{Z}$)**.** Let $a, b \in \mathbb{Z}$ with $b > 0$. Then, there exists unique integers q and r with $a = bq + r$, such that $0 \leq r < b$.

Remark 1.2. From theorem 1.1, we are sure that the remainder r when a is divided by m satisfies $0 \leq r < m$.

Notation 1.3. We write $a \pmod m$ to denote the remainder when a is divided by m.

We are now ready to state the problem state the problem.

Problem 1.4. Let b, x, m be integers with n non-negative and $m > 0$. Compute for $b^n \pmod m$.

The naïve approach to this problem is to solve

$$b, b^2, b^3, b^4, \ldots, b^n \pmod m.$$

However, this will take $O(n)$ multiplications[1]. However, we can do better. For simplicity, we first assume that n is a power of 2, say $n = 2^x$. We then make the trivial observation that

$$b^n \equiv b^{n/2} \cdot b^{n/2} \pmod m.$$

We have *divided* n into two parts. We can do it again. We have that $b^{n/2} \equiv b^{n/2^2} \cdot b^{n/2^2} mod n$. Continuing, we eventually reach $b^{n/2^{x-1}} = b^2 \equiv b \cdot b \pmod m$. This means, that to solve $b^n = b^{2^x} \pmod m$, we just need to solve

$$b, b^2, b^{2^2}, \ldots, b^{2^x} \pmod m.$$

[1] Very very informally speaking, $O(n)$ means that you will take around cn multiplications where c is some constant.

These can be solved by $O(x)$ squarings (or x multiplications). Take note that $x = \log_2 n$. And so, $O(\log n)$ multiplications is clearly an improvement over the naïve approach which neeeds $O(n)$ multiplications. However, we are not yet done. What if n is not a power of 2? What if it's odd? If we assume that n is odd, then we can write it as $n = 2k + 1$ where k is an integer. Thus, we have

$$b^n \equiv b^{2n+1} \equiv b^{2n} \cdot b \pmod{m}.$$

But take note that we can do the divide and conquer thingy with b^{2n} since it's equal to $b^n \cdot b^n$. Hence, just like the approach we just finished talking about, we will only ever need to compute for $b, b^2, b^{2^2}, \ldots \pmod{m}$. Hence, we will also need to make $O(\log n)$ multiplications. Finally, we state the algorithm in an organized manner:

Algorithm 1.5 (modular exponentiation)**.** `modpow(b, n, m)`
Input b, n, m integers with n non-negative and $m > 0$.
Output $b^n \pmod{m}$.

1. If n is 0, return $1 \pmod{m}$.
2. If n is 1, return $b \pmod{m}$.
3. If $n \geq 2$:
   (a). Solve for $x =$ `modpow(b, n/2, m)`. Here, / means integer division.
   (b). If n is even, return $x^2 \pmod{m}$.
   (c). If n is odd, return $x^2 \cdot b \pmod{m}$.

Implementation 1.6. Here is an implementation of algorithm 1.5.

```
modpow(b, n, m):
    if n == 0:
        return 1
    if n == 1:
        return b
    x = modpow(b, n/2, m);
    if n%2 == 0:
        return (x*x)%m
    else
        return (b*x*x)%m
```

1.2 Binary Search

Another generic problem-solving technique that uses the divide-and-conquer paradigm is the binary search method. We state the following problem:

Problem 1.7. Let $p(x) : \{0, 1, \ldots, n-1\} \to \{\texttt{true}, \texttt{false}\}$ be a function such that

$$p(i) = \begin{cases} \texttt{false} & \text{if } i < k \\ \texttt{true} & \text{if } i \geq k. \end{cases} \tag{1.8}$$

Given the array and a way to determine the answer to $p(j)$ for any j, determine k.

The naïve approach to this problem is to go through each element of the array from left to right and check if $p(x) =$`true`. This means that you will have to call p at most n times! Not only is this inefficient, but it does not utilize the fact that the special form of the function. Now, how do we abuse this special property of f? We make the simple observation that:

Observation 1.9. If $p(j)$ is `true`, then $k \leq j$ and if $p(j) =$`false`, then $j < k$.

This means that if we choose j such that x_j is the middle element of the array, we essentially cut our search space in half! And so, we have the following algorithm:

Algorithm 1.10 (binary search)**.** `binsearch(a, b, p)`
Input integers $a, b \in \{0, 1, \ldots, n-1\}$ and a function p satisfying 1.8.
Output the lowest index $a \leq k \leq b$ such that $p(k) =$`true`.

1. If $a > b$, then k does not exist.
2. If $a = b$, check if $p(a) =$ `true`.
   - If it is true, then return a.
   - If it is false, then k does not exist.
3. Determine an integer j such that $|j - a|$ and $|b - j|$ differ by at most 1.
4. Check if $p(x_j) =$`true`.
   - If it is true, then $a \leq k \leq j$. Return `binsearch(arr, a, j, p)`.
   - If it is false, then $j < k \leq b$. Return `binsearch(arr, j+1, b, p)`.

Now, let's turn this algorithm into code. Note that we can let `j = (a + b)/2`, this is the average of a and b. However, a nasty test case might force `a + b` to overflow. However, there is no need to worry because by the power of mathematics, we have that

$$j \quad = \quad \frac{a+b}{2} \quad = \quad \frac{2a}{2} + \frac{b-a}{2} \quad = \quad a + \frac{b-a}{2}.$$

Math has saved the day! We now conclude that the safer option is to take `j = a+(b-a)/2`. Indeed, if a and b are non-negative integer whose value is at most `MAX_INT`, then $b - a$ will also be at most `MAX_INT`. Unlike God's love, your computations will not be overflowing.

Remark 1.11. Algorithm 1.10 requires $O(\log n)$ evaluations of p.

Implementation 1.12. Here is an implementation of 1.10.

```
binsearch(lo, hi, p):
   while lo < hi:
      mid = lo + (hi - lo)/2
      if p(mid) == true:
         hi = mid
      else:
         lo = mid+1
   if p(lo) == true:
      return lo
   return no_answer
```

Example 1.13. As problem 1.7 is stated rather generally, here are a few example problems wherein you can apply this algorithm.

1. Given a sorted array of integers $x_0 \leq x_1 \leq \ldots \leq x_{n-1}$, find the smallest integer in the array whose value is at least k.

   Solution: Let $p(i) = (x_i \geq k)$. Note that this satisfies 1.8. Then the answer is $x_{\texttt{binsearch(0,n-1,p)}}$.

2. Given a sorted array of integers $x_0 \leq x_1 \leq \ldots \leq x_{n-1}$, determine if the integer k is on the list.

   Wrong solution: Let $p(i) = (x_i = k)$. This does not satify 1.8.

   Solution: Let $p(i) = (x_i \geq k)$. Note that this satisfies 1.8. Say yes if and only if $x_{\texttt{binsearch(0,n-1,p)}} = k$.

3. Given a sorted array of integers $x_0 \leq x_1 \leq \ldots \leq x_{n-1}$, find the largest index i such that $x_i = k$.

   Solution: Let $p(i) = (x_i > k)$. Note that this satisfies 1.8. Let $j = \texttt{binsearch(0,n-1,p)}$. If $j > 0$, return $j - 1$ if and only if $x_{j-1} = k$. Otherwise, k does not occur.

4. Given a dictionary with words listed in alphabetical order, determine if the word `banana` is there.

5. Given a list of (not necessarily sorted) list of integers $x_0, \ldots, x_{n-1}$ whose first k entries are composite numbers and the rest are prime, find the first prime on the list.

   Solution: Let $p(i) = (x_i$ is prime). Note that this satisfies 1.8. Then the answer is $x_{\texttt{binsearch(0,n-1,p)}}$. Note that evaluating p is costly in this case. If we used the naive approach, we would be doing it $O(n)$ times. Here, we do it $O(\log n)$ times.

6. Given a polygon X completely contained in the first quadrant of the 2D cartesian plane, find the line which divides X into 2 equal parts.

   Solution: Let $p(m) =$ (the area of X on the 'left side' of the line $y = mx$ is $\geq k$). Let a and b be such that X is completely on the right side of $y = ax$ and on the left side of $y = bx$. Return $m_{\texttt{binsearch(a,b,p)}}$. Note that the domain of p is not the finite set $\{0, 1, \ldots, n-1\}$ anymore but instead $[a, b] \subseteq \mathbb{R}$. Can you formulate a new version of 1.8 that adapts to this problem?

1.3 Merge Sort

One of the fastest sorting algorithms follows the divide-and-conquer paradigm. It is more commonly known as merge sort. But before we can state the problem of sorting in full generality, we define the notion of a totally-ordered set.

Definition 1.14. A totally ordered set $(S, \leq)$ is a set S equipped with a comparison function $\leq$ on X such that for any $a, b, c \in S$, the following conditions hold:

1. **reflexivity**: $a \leq a$
2. **antisymmetry**: if $a \leq b$ and $b \leq a$, then $a = b$
3. **transitivity**: if $a \leq b$ and $b \leq c$, then $a \leq c$
4. **trichotomy**: exactly one of the following is true: $a \leq b$ or $b \leq a$

Example 1.15. If $(X, \leq)$ is a totally-ordered set, then we can define a comparison function $\geq$ on X such that $x \geq y$ if and only if $y \leq x$. Then $(X, \geq)$ is a totally-ordered set.

Example 1.16. If $(X, \leq)$ and $(Y, \leq)$ are totally-ordered sets, then we can define a comparison function $\leq$ on $X \times Y$ as follows: For any $a = (x_0, y_0), b = (x_1, y_1) \in X \times Y$, we say that $a \leq b$ if one of the following conditions is satisfied:

1. $x_0 \leq x_1$, or

2. $x_0 = x_1$ and $y_0 \leq y_1$

Then $(X \times Y, \leq)$ is a totally-ordered set.

Example 1.17. Consider the set

$$C = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{J}, \mathtt{K}, \mathtt{L}, \mathtt{M}, \mathtt{N}, \mathtt{O}, \mathtt{P}, \mathtt{Q}, \mathtt{R}, \mathtt{S}, \mathtt{T}, \mathtt{U}, \mathtt{V}, \mathtt{W}, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\}.$$

Let $f : C \to \{65, \ldots, 90\}$ such that $f(\mathtt{A}) = 65$, $f(\mathtt{B}) = 66$, etc. Define $\leq$ to be a comparision function on C such that for each $a, b \in C$, $a \leq b$ if and only if $f(a) < f(b)$ (as integers).

Example 1.18. Let S be a set and consider $(\mathcal{P}(S), \subseteq)$, where $\mathcal{P}(S)$ is the set of subsets of S and $\subseteq$ is the usual subset comparison function. Note that this is **not** a totally-ordered set. To show this, consider the set of $S = \{0, 1\}$. Note that $\{0\}$ and $\{1\}$ are subsets of S (and thus elements of $\mathcal{P}(S)$). However, observe that $\{0\} \not\subseteq \{1\}$ and $\{1\} \not\subseteq \{0\}$ and this violates the trichotomy property in definition 1.14. Indeed, it is **not** a totally-ordered set. However, such sets which do not necessarily have the trichotomy property but satisfy the three other properties are called partially-ordered sets.

Definition 1.19. If $(X, \leq)$ is a totally ordered set in which X is finite, then there exists s and t such that for all $x \in X$, we have that

$$s \leq x \qquad \text{and} \qquad x \leq t.$$

We call s the smallest element of X and t to be the largest element of X. We denote s by $\min(X)$ and t by $\max(X)$.

Notation 1.20. When the context is clear, we usually drop the $\leq$ and write just the name X of the set when we talk about a totally-ordered set $(X, \leq)$.

Problem 1.21. Let $(X, \leq)$ be a totally-ordered set. Let $x_0, \ldots, x_{n-1} \in X$. Find a one-to-one function $s : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ such that

$$x_{s(0)} \leq x_{s(1)} \leq \cdots \leq x_{s(n-1)}.$$

The **divide** step of the merge sort algorithm, as usual, involves splitting the array into two parts. The much more interesting step is the **combine** step (or the merge step). Here, we assume that we already have two sorted arrays S_1 and S_2:

S_1	S_2

Suppose we want S to be the final sorted array. As both subarrays S_1 and S_2 are sorted, then $\min(S)$ will either be $\min(S_1)$ or $\min(S_2)$, depending on which one is smaller[2]. After figuring out the first element of S (i.e. the smallest), we remove it from its respective subarray. Now, we take the minimum between $\min(S_1)$ and $\min(S_2$ to find the second (smallest) element of S. And so on.

Algorithm 1.22 (merge sort)**.** `mergesort(a, b, f)` Fix $x_0, \ldots, x_{n-1} \in X$, where X is a totally-ordered set.
Input $a, b \in \{0, \ldots, \mathrm{n}-1\}$
Output a one-to-one function $s(x) : \{a, \ldots, b\} \to \{a, \ldots, b\}$ such that $x_{s(i)} \leq x_{s(j)}$ if and only if $i \leq j$.

1. If $a = b$, then let $s(a) = a$.
2. Determine an integer j such that $|j - a|$ and $|b - j|$ differ by at most 1.
3. Let s_1 = `mergesort(a, j)` and s_2 = `mergesort(j+1, b)`.

[2]We break ties by comparing the original index, which are both integers. In this case, we take $\min(S_1)$

4. Let $i_1 = a$, $i_2 = j + 1$.
5. For $k = a, a + 1, \ldots, b$:
   - If $i_2 > b$, set $s(k) = s(i_1)$ and increment i_1 by 1.
   - If $i_1 > j$, set $s(k) = s(i_2)$ and increment i_2 by 1.
   - If $x_{s_1(i_1)} \leq x_{s_2(i_2)}$, then set $s(k) = s(i_1)$ and increment i_1 by 1.
   - Otherwise, set $s(k) = s(i_1)$ and increment i_2 by 1.
6. Return s.

Remark 1.23. Algorithm 1.22 requires $O(n \log n)$ comparisons. This is easy to see if $n = 2^x$. because you will have to call the function with an array of size n one time, with an array of size $n/2$ two times, an array of $n/4$ four times, and so on until you arrive at an array of $n/2^x = 1$ and you have to deal with n elements at each level. This means that the depth of the recursion will be $\log n$. Moreover, at each depth you will need to compare n times[3]

Implementation 1.24. Here is an implementation of 1.22.

```
mergesort(lo, hi, s):
   if lo == hi:
      return [ lo ];
   mid = lo + (hi - lo)/2
   s1 = mergesort(lo, mid)
   s2 = mergesort(mid+1, hi)
   i1 = lo
   i2 = mid+1
   arr = [ ];
   for k = lo, ..., hi:
      if i2 > hi:
         arr.append( s1[i1] )
         i1++
      if i1 > mid:
         arr.append( s2[i2] )
         i2++
      if x[s1[i1]] <= x[s2[i2]]:
         arr.append( s1[i1] )
      else:
         arr.append( s2[i2] )
```

1.4 Longest Increasing Subseqence

Suppose $a_1, a_2, \ldots, a_n$ are from a totally-ordered set $(X, \leq)$. We call this a finite sequence S in X of length n. A subsequence of S is obtained by removing some (possibly none, possibly all) of the terms of S. For people who like formal definitions:

Definition 1.25. Let $S = (a_1, a_2, \ldots, a_n)$ be a sequence. Let ℓ be a non-negative integer such that $\ell \leq n$. Consider $i_1, \ldots, i_\ell$ such that

$$1 \leq i_1 < i_2 < \cdots < i_\ell \leq n.$$

[3]For example, since you will call the function 2^i times with an input array of size $n/2^i$, this batch of calls will require about $2^i \cdot n/2^i = n$ comparisons.

Then,

$$a_{i_1}, a_{i_2}, \ldots, a_{i_\ell}$$

is a subsequence of S. Note that if $\ell = 0$, we end up with an empty sequence. For our purposes, we consider this a subsequence of S.

Optional Exercise 1.26. How many subsequences does a sequence of length n have? Assume that all the terms are distinct for simplicity.

We also define the following types of sequences:

Definition 1.27. Let $S = (a_1, a_2, \ldots, a_n)$ be a sequence in a totally-ordered set $(X, \leq)$. We write $a < b$ to denote that $a \leq b$ but $a \neq b$.

- If
$$a_1 < a_2 < \cdots < a_n$$
then S is said to be a (strictly) increasing sequence.
- If
$$a_1 \leq a_2 \leq \cdots \leq a_n$$
then S is said to be a non-decreasing sequence.
- If
$$a_1 > a_2 > \cdots > a_n$$
then S is said to be a (strictly) decreasing sequence.
- If
$$a_1 \geq a_2 \geq \cdots \geq a_n$$
then S is said to be a non-increasing sequence.

We are now ready to pose the main problem of this section:

Problem 1.28. Given a sequence S in a totally-ordered set X, find the length of the longest increasing subsequence of S.

Optional Exercise 1.29. One naïve solution to problem 1.28 has complexity $O(2^n)$. Figure it out.

1.4.1 An $O(n^2)$ solution

One solution to problem 1.28 is by using dynamic programming. Here's the solution! Spoiler alert. Let $\ell(j)$ be the length of the longest increasing subsequence ending in a_j. Now, if we know $\ell(i)$ for each $i < j$, then we will be able to solve $\ell(j)$ as follows:

$$\ell(j) = \max(S_j) \tag{1.30}$$

where

$$S_j = \{\ell(i) + 1 : i < j \text{ and } a_i < a_j\} \cup \{1\}.$$

This is because if $a_k < a_m$, then we can add a_m to the longest increasing subsequence ending in a_k. With this idea, we are now ready to discuss the algorithm.

Algorithm 1.31 (longest increasing subsequence)**.** `lisdp(a, n)`

Input a sequence $S = (a_1, \ldots, a_n)$ in a totally-ordered set X of length n

Output a function ℓ such that $\ell(i)$ is the length of the longest increasing subsequence whose last term is a_i, a function $p(i)$ such that $a_{p(i)}$ is the penultimate term of the longest increasing subsequence whose last term is a_i, and a sequence $(s_1, \ldots, s_m)$, a longest increasing subsequence of S.

- Set $\ell(1) = 1$ and $p(i) = -1$.
- Set $m = 1$;
- For each $j = 2, \ldots, n$:
  - Set $\ell(j) = 1$.
  - For each $i = 1, 2, \ldots, j-1$:
    * If $a_i < a_j$ and $\ell(i) + 1 > \ell(j)$, then let $\ell(j) = \ell(i) + 1$ and set $p(j) = i$. This solves 1.30.
  - If $\ell(j) > \ell(m)$, then set $m = j$.
- Let $t = \ell(m)$.
- Let $k = m$.
- While $k \neq -1$:
  - Let $s_t = a_k$.
  - Decrease t by 1.
  - Let $k = p(k)$.

Implementation 1.32. Here is an implementation of 1.31.

```
lisdp(a, n):
   l[1] = 1
   p[1] = -1
   max = 1;
   for j = 2, ..., n:
      l[j] = 1
      for i = 1, ..., j-1:
         if a[i] < a[j] and l[i]+1 > l[j]:
            l[j] = l[i]+1
            p[j] = i
      if l[j] > l[max]:
         max = j
   t = l[m]
   k = m
   while k != -1:
      s[t] = a[k]
      t--
      k = p[k]
```

Note that there are two nested for loops. This means that whatever's done in the innermost loop will be repeated $n(n+1)/2 = O(n^2)$ times. Finally, constructing the sequence will take at most $n = O(n)$ operations. Hence, much of the computations in the algorithm are done in the nested for loops and hence the algorithm runs in $O(n^2)$ time.

1.4.2 A faster solution

While the $O(n^2)$ dynamic programming solution is already impressive, there is an even faster solution. In this new solution, we replace the inner for-loop by something that takes $O(\log n)$ time. In algorithm 1.31, we are essentially storing all longest increasing subsequences that end in each of the a_i. Hence, we may have been storing more than one longest increasing subsequences of length r, for example. Our new idea is that we instead store one longest increasing subsequence of length r – the one whose last term is minimal.

Definition 1.33. We define $S_{j,r}$ to be the longest increasing subsequence of $a_1, a_2, \ldots, a_j$ of length r whose last term is minimal. We denote by $L_{j,r}$ to be the last term of $S_{j,r}$.

Claim 1.34. For a fixed j, we have $L_{j,1} \leq L_{j,2} \leq \ldots \leq L_{j,r_j}$ (where $r_j = \max\{\ell(1), \ldots, \ell(j)\}$).

Proof. We prove by contradiction. Suppose there exists $a < b$ such that $L_{j,a} > L_{j,b}$. Exercise: Find a contradiction. □

Now, if we have the sequences $S_{j-1,1}, \ldots, S_{j-1,r}$, whose last terms are $L_{j-1,1}, \ldots, L_{j-1,r}$, then we know that we can replace with a_j the largest $L_{j-1,t}$ such that $a_j < L_{j-1,t}$. If this does not exist, then we append a_j to the longest increasing subsequence we have so far. We acknowledge that this is a bit too much to take in, and so here is an animation available on Wikipedia demonstrating what we have just said. And so, the new and faster algorithm goes as follows:

Algorithm 1.35 (longest increasing subsequence)**.** `lis(a, n)`
Input a sequence $S = (a_1, \ldots, a_n)$ in a totally-ordered set X of length n
Output a function ℓ such that $\ell(i)$ is the length of the longest increasing subsequence whose last term is a_i, a function $p(i)$ such that $a_{p(i)}$ is the penultimate term of the longest increasing subsequence whose last term is a_i, and a sequence $(s_1, \ldots, s_m)$, a longest increasing subsequence of S.

- Set ~~$\ell(1) = 1$ and~~ $p(i) = -1$.
- ~~Set $m = 1$~~;
- Let $\mathcal{L}$ be a list (of indices). Add 1 to this list.
- For each $j = 2, \ldots, n$:
  - Use binary search to find the largest index r such that $a_j \leq a_{\mathcal{L}(r)}$.
  - If r exists:
    * Set $p(j) = p(\mathcal{L}(r))$.
    * Set $\mathcal{L}(r) = j$.
  - If r does not exist:
    * Set $p(j) = \mathcal{L}(m)$, where m is the length of $\mathcal{L}$.
    * Set Append j to $\mathcal{L}$.
- ~~Let $t = \ell(m)$.~~
- Let $k = m$, where m is the length of $\mathcal{L}$.
- While $k \neq -1$:
  - Let $s_t = a_k$.

- Decrease t by 1.
- Let $k = p(k)$.

Implementation 1.36. Here is an implementation of 1.35.

```
lis(a, n):
   p[1] = -1
   L = [1]
   for j = 2, ..., n:
      m = length(L)
      r = binsearch(1, m, p)-1   // p(i) = ( L[i] < a[j] )  You can implement your own binary search if you want.
      if r > 0:
         p[j] = p[L[r]]
         L[r] = j
      else:
         p[j] = L[m]
         L.append(j)
   k = length(L)
   while k != -1:
      s[t] = a[k]
      t--
      k = p[k]
```

Chapter 2

Number Theory

2.1 Modulo

Definition 2.1 (divisibility)**.** We say that a non-zero integer b divides an integer a (written $b|a$) if there exists $q \in \mathbb{Z}$ such that $a = bq$. In this case, we say that b is a divisor of a.

Notation 2.2 (congruence modulo m)**.** We write

$$a \equiv b \pmod{m}$$

to mean $m|(a - b)$. We read this as 'a is equivalent to b modulo m'.

Definition 2.3. Let m be a positive integer. We denote by $\overline{a}$ the set

$$\overline{a} = \{b \in \mathbb{Z} : a \equiv b \,(\text{mod } m)\}.$$

We define $\mathbb{Z}_m$ to be the set

$$\mathbb{Z}_m := \{\overline{0}, \overline{1}, \ldots, \overline{m-1}\}.$$

Definition 2.4 (addition on $\mathbb{Z}_m$)**.** Let m be a positive integer. We define addition $+_m$ on the elements of $\mathbb{Z}_m$ as:

$$\overline{a} +_m \overline{b} = \overline{a + b}.$$

Theorem 2.5. Addition on $\mathbb{Z}_m$ shares some similar properties to the normal addition $+$ on integers.

1. It is closed.

   This means that if you add any two elements of $\mathbb{Z}_m$, the result will be in $\mathbb{Z}_m$. Indeed:

   $$\overline{a} +_m \overline{b} = \overline{a + b} = \overline{r}$$

   where r is an integer $0 \leq r < m$ such that $a + b = qm + r$, as in theorem 1.1.

2. It has an additive identity, $\overline{0}$. We sometimes call this the zero element.

   This means that $\overline{a} + \overline{0} = \overline{a}$ for any element $\overline{a} \in \mathbb{Z}_m$.

3. Any element $\overline{a}$ has an additive inverse $\overline{-a} = \overline{m - a}$.

   This means that $\overline{a} +_m \overline{-a}$ is equal to the additive identity for any element $\overline{a} \in \mathbb{Z}_m$.

Definition 2.6 (multiplication on $\mathbb{Z}_m$)**.** Let m be a positive integer. We define addition $\cdot_m$ on the elements of $\mathbb{Z}_m$ as:

$$\overline{a} \cdot_m \overline{b} = \overline{a \cdot b}.$$

Theorem 2.7. Multiplication on $\mathbb{Z}_m$ shares some similar properties to the normal multiplication $\cdot$ on integers.

1. It is closed.
2. It has an multiplicative identity, $\overline{1}$.

   This means that $\overline{a} \cdot_m \overline{1} = \overline{a}$ for any non-zero element $\overline{a}$.

Notation 2.8. Instead of writing $\overline{a} +_m \overline{b}$ and $\overline{a} \cdot_m \overline{b}$, we write

$$a + b \pmod{m} \qquad \text{and} \qquad ab \pmod{m}$$

respectively.

Example 2.9. We prove the divisibility rule by 9: an integer x is divisible by 9 if and only if the sum of its digits is divisible by 9.

Proof. Let $x = a_d \cdot 10^d + a_{d-1} \cdot 10^{d-1} + \ldots + a_1 \cdot 10 + a_0$. Note that $10 \equiv 1 \pmod 9$. Hence, $10^n \equiv 1 \pmod 9$ for any non-negative integer n. Thus,

$$x \quad = \quad a_d \cdot 10^d + a_{d-1} \cdot 10^{d-1} + \ldots + a_1 \cdot 10 + a_0 \quad \equiv \quad a_d + a_{d-1} + \ldots + a_1 + a_0 \pmod 9.$$

This proves the claim. □

2.2 Prime Factorization

There are many algorithms to determine prime factors of numbers. Most of these involve choosing random integers along the way, which makes them unappealing for programming contests. And so, for most problems in programming contests, only prime factorizations of small numbers are needed to be factored in order to complete the algorithm. As these algorithms are very much standard, they will probably be used in conjuction with other algorithms. Hence, it is vital to know these basic number-theoretical algorithms.

Definition 2.10. A prime number p is an integer greater than 1 that has no positive integer divisors other than 1 and p itself.

Algorithm 2.11 (Sieve of Eratosthenes)**.** `sieve(N)`
Input an integer N
Output an array `primeBa` of length N such that for any $n \leq N$, `primeBa[n] = 1` if n is prime and `primeBa[n] = 0`, otherwise.

1. Initialize the elements of `primeBa` to 1.
2. Let `primeBa[1] = 0`.
3. For each $p = 2, 3, 4, 5, \ldots$:
   - If `primeBa[p]` is equal to 0, continue to the next value for p.
   - For each integer $k > 1$ such that $kp \leq N$, set `primeBa[k*p] = 0`.
4. Return `primeBa`.

Remark 2.12. The running time for this algorithm is $O(N \log\log N)$. In the algorithm, we make around

$$\frac{N}{2} + \frac{N}{3} + \frac{N}{5} + \frac{N}{7} + \frac{N}{11} + \dots$$

operations. And we use the fact (which we will not prove) that

$$\sum_{p,\text{ prime},\, p \le N} \frac{1}{p}$$

is approximately equal to $\log\log N$ when N is very large.

Implementation 2.13. Here is an implementation of algorithm 2.11.

```
sieve(N):
   primeBa[1] = 0
   for k = 2, 3, 4, 5, ..., N
      primeBa[k] = 1
   for p = 2, 3, 4, 5, ..., N
      if primeBa[p] == 0:
         continue
      for i = 2, 3, 4, 5, ...
         if i*p > N:
            break
         primeBa[i*p] = 0
   return primeBa
```

Theorem 2.14 (fundamental theorem of arithmetic)**.** Any positive integer $n > 1$ can be expressed as

$$n = p_1^{e_1} p_2^{e_2} \cdots p_t^{e_t} \tag{2.15}$$

where the p_i are distinct prime factors and e_i are positive integers.

Remark 2.16. We call the expression in 2.15 the prime factorization of n.

Remark 2.17. Algorithm 2.11 can be modified to find the prime factorization of all integers below N.

Definition 2.18 (greatest common divisor)**.** The greatest common divisor of two integers a and b is the largest integer d such that $d|a$ and $d|b$.

Theorem 2.19. Let

$$a = p_1^{e_1} p_2^{e_2} \cdots p_t^{e_t} \qquad \text{and} \qquad b = p_1^{f_1} p_2^{f_2} \cdots p_t^{f_t}$$

be the prime factorization of two positive integers a and b (some of the e_i and f_i may be zero). The greatest common divisor, $\gcd(a, b)$, of a and b is:

$$\gcd(a, b) = p_1^{\max\{e_1, f_1\}} p_2^{\max\{e_2, f_2\}} \cdots p_t^{\max\{e_t, f_t\}}.$$

Theorem 2.20. Here are some properties concerning the greatest common divisor:

1. $\gcd(a, 0) = 0$.
2. If $b < a$, then $\gcd(a, b) = \gcd(b, a - b)$.
3. If $b < a$, then $\gcd(a, b) = \gcd(b, r)$ for r such that $r \equiv a \pmod{b}$ and $0 \le r < b$.

Optional Exercise 2.21. Prove theorem 2.20. Hint: Use definition 2.1 when appropriate.

From theorem 2.20, we come up with this simple algorithm to find the greatest common divisor of two integers a and b:

Algorithm 2.22 (Euclidean algorithm)**.** `gcd(a, b)`

Input two non-negative integers a and b

Output $d = \gcd(a, b)$

1. If $a < b$, return `gcd(b, a)`.
2. If $b = 0$, return a.
3. Return `gcd(a%b, b)`.

Remark 2.23. The complexity of algorithm 2.22 is $O(\log n)$.

Implementation 2.24. Here is an implementation of algorithm 2.22.

```
gcd(a, b):
   if a < b:
      return gcd(b, a)
   if b == 0:
      return a
   return gcd(a%b, b)
```

Chapter 3

Combinatorics

Problem 3.1. How many ways can one arrange n distinct objects in a row of n?

There are

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1$$

ways to arrange n distinct objects in a row. You have n choices for the first one. Once you've set that, you will have $n-1$ choices for the next and then $n-2$ and so on until you have 1 object left for the nth slot.

Problem 3.2. How many ways can one arrange n distinct objects in a row of k where $k \leq n$?

You will have n choices for the first slot, $n-1$ for the second, all the way down to the kth slot which will have $n-k+1$. Hence, you will have

$$n \cdot (n-1) \cdot n-2 \cdot \cdots \cdot (n-k+1).$$

You can also get the same answer by first considering a solution for the first problem, and then taking note that the order of the last $n-k$ elements does not matter (because they will not be in the row anyway). So you can jumble them up in whatever order you like. And from the answer to the problem 3.1, there are $(n-k)!$ ways to jumble them up. So with the original answer $n!$, you treat the $(n-k)!$ arrangements of the last $n-k$ items as 1. And so, you get the same answer

$$\frac{n!}{(n-k)!} = n \cdot (n-1) \cdot n-2 \cdot \cdots \cdot (n-k+1).$$

Problem 3.3. How many ways can one choose k different objects from n distinct objects where $k \leq n$?

Starting from the answer to problem 3.2, we are now allowed to jumble the first k objects as well. That means the $k!$ different arrangements of the first k items will be treated as 1. Hence, we end up with the formula:

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

Remark 3.4. Zero factorial is equal to the empty product. That is $0! = 1$.

While the solutions presented in the above problems are intuitive, we have another way to approach these kinds of problems. Take for example problem 3.3. Let's abbreviate problem 3.3 as $\mathcal{P}(n,k)$. This denotes the problem of choosing k different objects from n distinct objects. If we had a solution for $\mathcal{P}(n-1,k-1)$, then we can add an nth element as a chosen one. We could also take a solution for $\mathcal{P}(n-1,k)$. In this case, we can add an nth element as an element which is not chosen (since the solution for $\mathcal{P}(n-1,k)$ already has k chosen elements). Hence, we have the relation

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}. \tag{3.5}$$

This value is called the binomial coefficient. These numbers arise in different contexts. For example, the polynomial obtained by raising $x + 1$ to the nth power will be equal to

$$(x+1)^n \quad = \quad x^n + \binom{n}{n-1}x^{n-1} + \binom{n}{n-2}x^{n-2} + \ldots + \binom{n}{2}x^2 + \binom{n}{1}x + \binom{n}{0}.$$

Remark 3.6. Here are some properties of the binomial coefficient.

1. For $0 \leq k \leq n$, we have $\binom{n}{k} = \binom{n}{n-k}$.
2. For $0 < k < n$, we have $\binom{n}{k}$ is divisble by n.

Optional Exercise 3.7. What is the remainder when $42^{69} + 26^{69}$ is divided by 69?

Optional Exercise 3.8. There are also a lot of other variations of counting problems. Here are some of them. Try using the answers to the previous problems, or use a similar technique to arrive at the answer.

1. How many distinct strings can you obtain by rearranging the letters of the string `BEBEBIBIGURL`?
2. How many ways can you seat yourself and $N - 1$ other people in a row if you insist on sitting with your one true love?
3. How many ways can you seat yourself and $N - 1$ other people in a row if your one true love insists on sitting at least one seat away from you?
4. Let S be a positive integer. How many positive integer solutions does $x_1 + x_2 + \ldots + x_n = S$ have?
5. Let S be a positive integer. How many non-negative integer solutions does $x_1 + x_2 + \ldots + x_n = S$ have?
6. How many ways can you assign n different pigeons in k different holes?
7. How many ways can you rearrange n pairs of (and) in a string such that each (is paired up with a) somewhere on its right?

Remark 3.9. One can implement a function which returns the binomial coefficient by means of a recursive function. If memory allows, one can do better by remembering all the results of the computations to avoid solving for the same thing twice.

Chapter 4

Exercises

Send an email **on or before 11:59PM of 2 April 2017 (Sunday)** to training@noi.ph containing (at least) the following:

- Your Codeforces, UVa, HackerRank, and ProjectEuler usernames.
- Links to the source code you submitted for the required Codeforces problems.
- Code (attached) used for the required ProjectEuler and UVa problems.
- Text, PDF or photo (attached) of all required proofs.

4.1 Required Exercises

1. Proof of 1.34
2. CF 215A: Points on Line
3. Project Euler 133: Repunit nonfactors
4. UVa 612: DNA Sorting
5. UVa 10113: Exchange Rates
6. UVa 10534: Wavio Sequence
7. UVa 10665: Contemplation! Algebra
8. UVa 13083: Yet Another GCDSUM
9. At least one of:
   (a). UVa 10247: Complete Tree Labeling (BigInt)
   (b). HackerRank Mirror : Complete Tree Labeling (Mod)

4.2 Optional Exercises

- UVa 11378: Bey Battle
- CF 294C: Shaass and Lights
- CF 327C: Magic Five
- CF 577B: Modulo Sum
- CF 272D: Dima and Two Sequences
- Project Euler 147: Rectangles in cross-hatched grids
- UVa 10229: Modular Fibonacci
- UVa 12192: Grapevine