NYU

CSCI-UA.0480-004
Algorithmic Problem Solving

Lecture 11
Divide and Conquer

Bowen Yu
Spring 2018

HW

- HW6
  - Union find
  - Due tomorrow
- HW7 released
  - Complete search, greedy I
  - Due on Sunday
- HW8 to be released
  - Complete search, greedy II
  - Due on next Wednesday

Python Power

- A test on modular arithmetic: compute a^b mod P.
- Since a is large, we can first mod a by P.
    - Treat a as a decimal string.
    - No need to use big integer.
- Then we multiply “a mod P” b times with modulus P to obtain the answer.
- Careful with 32-bit overflow in multiplication.
- Time: $O(|a| + b)$

Span Queries

- Use a BBST to maintain the numbers (add, delete).
- Retrieve the leftmost node in the BBST as the min, and the rightmost node in the BBST as the max, in order to compute the span.
- Time: O(qlogq)

- Using the library’s heap may have a issue with deletion.
  - PriorityQueue.remove is linear!

Bracket Sequence II

- Similar to homework: If the sequence can be fixed by inserting one bracket, then it must be of the form:
  - $l_1 l_2 l_3 \ldots l_k x r_k \ldots r_3 r_2 r_1$
    - l_i is a left bracket and r_i is a right bracket, and l_i matches r_i
    - x is any bracket
- Use a stack to check validness of the sequence. If there is a mismatched bracket, also push it to the stack.
- Finally treat the stack as an array and check if it has the pattern above.

Students in a Row II

Solution 1

- Build a doubly linked list for the students. “ml/mr x” is to delete x and then insert at the head/tail of the list.
- Time: O(n + q)

Solution 2

- Given each student a position value. Initially student x has position x. Maintain the positions in a BBST.
- Record the leftmost and rightmost positions: left=1, right=n.
- For ml, assign “--left” to the student’s position. For mr, assign “++right” to the student’s position.
- For l/r query, find the student in the BBST, and retrieve its predecessor and successor node in the BBST.
- Time: O((n + q)logn)

Image Convolution

Store the image and pattern as bitmasks. First let's assume the pattern has no question marks. Suppose each block has B bits. e.g. for B = 8:

```
image   = 00110100 | 10111011 | 10101
pattern = 01101000 | 01
           0110100 | 001
            011010 | 0001
```

Shift the pattern and compare the bits. We can compare the image against the pattern B bits at a time. The time complexity is O(nmab/B). This 1/B speed boost is required to pass the large subtask.

image   = 00**110100** | **10** 111011 | 10101
pattern = (011010   00)01

How to compare when the bitmasks are not aligned? Bit manipulation.

image   x = 00<u>110100</u>, y = <u>10</u>111011
pattern z = 01101000

We want the last 6 bits of x and highest 2 bits of y. Use bit manipulation (<<, >>, &) to obtain an integer that is <u>11010010</u>, then we can compare this against z.

How to handle positions that must be ones?zeroes?
How to handle question marks?

Create a set of pattern bitmasks M1 that has every ‘#’ to be 1.
Create a set of pattern bitmasks M0 that has every ‘.’ to be 1.
Ignore the question marks completely.

Then a match satisfies: M1 & image = M1, M0 & image = 0

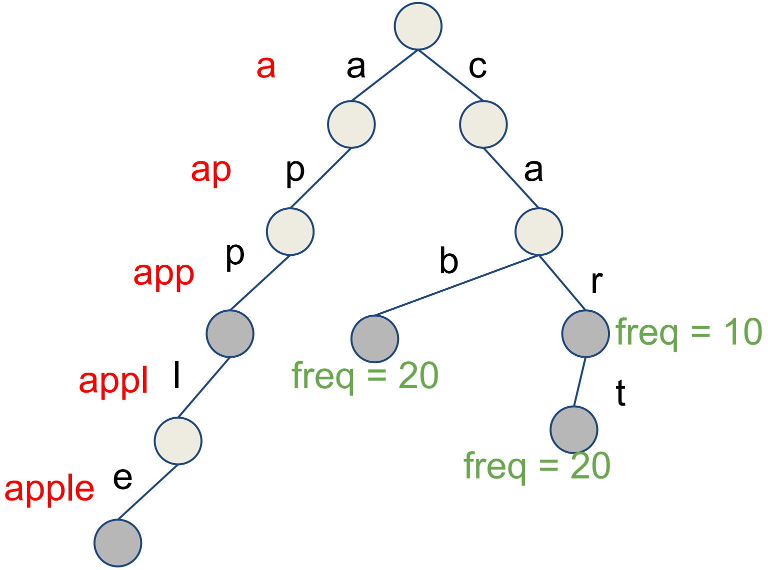
- Using the library's bitset
  - Totally fine.
  - But make sure that you also know how to manipulate integer bitmasks yourselves. We will be doing this a lot on some future topics.
    - e.g. In subset enumeration we typically work with a bitmask with ~20 bits. It is an overkill to use bitset. Using a 32-bit integer can be faster.

- Using the library's big integer.
  - Kind of weird.

Auto Completion

- Solution to the easy subtask has been directly covered lecture.
- For each Trie node X, we precompute to which Trie node it auto completes to, say ans(X). Then
  - ans(X) =
    - ans(Y) | Y is a child of X, ans(Y).string has max frequency and is smallest, or
    - Y | Y is a child of X, Y.string has max frequency and is smallest

- For a Trie node X, X.string is an index, not the string itself. Otherwise we exhaust the memory!
  - $O(L^2)$ space, L is 1M, the total number of characters.
- On a tie of max frequence, there is no need to compare the original strings.
  - “cab” has same max frequency as “cart”.
  - The answer is the word following edge “b” because “b” < “r”.



How to handle multiple tab presses?

- Pressing tab once is essentially jumping from a node X in the Trie to the node X completes to.
- Whenever we see a tab, we perform this jump. Since each node X knows which node it should jump to (precomputed), each jump takes constant time.

- Time
  - O(|strokes|) per query
  - O(L) in total, where L is the total number of characters in the input (2M).

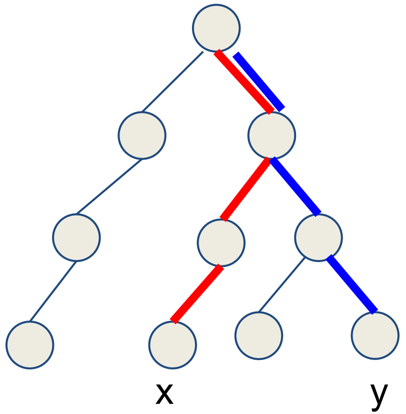
Max XOR on a Tree

- The small subtask gives an array, and is exactly the max XOR subarray problem covered by the lecture.
- Compute xor(i) = a[i] ^ a[i-1] ^ a[i-2] … a[1]
- Find the pair (i, j) where xor(i) ^ xor(j-1) is maximum using a Trie.

How to extend it to handle a path on the tree?

- Make the tree rooted.
- let xor(x) be the XOR of edge weights on the path from x to the root.
- For a pair of nodes x, y, the XOR of path (x, y) is then xor(x) ^ xor(y).
- Find the pair (x, y) using Trie.

- Time: O(30n), 30 is the number of bits needed to store 10^9.



You drive a car along a road. Your car initially has a full tank of capacity C. You start at x=0 and want to reach x=L. There are $N<=10^5$ gas stations on the road. The i-th station is at Xi (0<Xi<L) and sells gas for Pi dollars per gallon. At each gas station, you can refill any amount of gas till a full tank. Determine the minimum total cost to reach x=L, assuming it costs 1 gallon of fuel to drive 1 unit of distance.

Sample:

C = 4, L = 6
X1 = 1, P1 = 3
X2 = 2, P2 = 2
X3 = 5, P3 = 1

Drive to X2, fuel 4->2
At X2, buy 1 gallon, fuel 2->3
Drive to X3, fuel 3->0
At X3, buy 1 gallon, fuel 0->1
Drive to L=6, fuel 1->0

Solution

- If we can find a station that has cheaper gas and within the range of a full tank's distance
  - If with the remaining fuel we can drive to it, then drive to it
  - Otherwise, buy just an amount of fuel so that we can reach that station with zero fuel left
- If we cannot find such a cheaper station, then refill to a full tank at the current station. Then drive to the next station.

Proof is implied by the greedy decision.

Straightforward implementation takes $O(N^2)$. Can be optimized to $O(N)$.

Divide and Conquer (D&C)

- Divide the problem into smaller subproblems of a same structure and solve each subproblem
- Merge the answers to the subproblems to form the answer to the original problem.
- Many data structures use D&C.

- **Merge Sort**
  - Split the array into two equal halves. Merge sort each half. Then merge the sorted half arrays.
  - Merging two sorted arrays takes linear time. We repeatedly take the smaller of the two arrays' heads.
  - $T(n) = 2T(n/2) + O(n) = O(n\log n)$
- **Quick Sort**
  - Choose a pivot element. Place the elements smaller than the pivot on the left, and the elements larger than (or equal to) the pivot element on the right. This rearrangement takes O(n).
  - Quick sort the left and right groups individually.
  - Ideally, the pivot splits the array into two equal halves. $T(n) = 2T(n/2) + O(n) = O(n\log n)$
    - But if we choose a bad pivot, this can downgrades to $O(n^2)$

Master Theorem

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Master Theorem (Intuitively)

$$T(n) = aT(n/b) + f(n)$$

- We want to compare $O(n^{\log_b a})$ and $f(n)$
- If one grows faster than the other, then the total complexity is that function.
- If both grows equally fast, then it's $O(n^{\log_b a} \log n)$

$T(n) = 2T(n/2) + O(1) \Rightarrow O(n)$

$T(n) = 2T(n/2) + O(n^2) \Rightarrow O(n^2)$

$T(n) = 3T(n/2) + O(n) \Rightarrow O(n\text{^}\log_2 3)$

$T(n) = 4T(n/2) + O(n^2) \Rightarrow O(n^2 \log n)$

$T(n) = T(2n/3) + O(1) \Rightarrow O(\log n)$

We want to compute (a^b mod P), where $P = 10^9 + 7$, $0 <= a, b <= 10^9$.

- If b is small (e.g. $<= 10^6$), we can just multiply b times. But for $b=10^9$, O(b) will TLE.

Solution

- We can compute a^1, a^2, a^4, a^8 and so on in log(b) time.
- Then we can take the powers matching b's binary representation.
  - e.g. $b = 5 = (101)_2$, then $a^b = a^1 * a^4$

Logarithm Power Implementation

```
int modPower(int a, int b, int P) {
  int ans = 1;
  while (b) {
    if (b & 1) ans = (long long)ans * a % P;
    b >>= 1;
    a = (long long)a * a % P;
  }
  return ans;
}
```

- The while loop is checking each bit of b, from the least significant to the most significant.
- After checking each bit, we square a, so that a becomes a^2, a^4, a^8, and so on.

We want to use 32-bit integers without cast to 64-bit to compute (a * b mod P), where $P = 10^9 + 7$, $0 <= a, b <= 10^9$.

```
int modMultiply(int a, int b, int P){
  int ans = 0;
  while (b) {
    if (b & 1) ans = (ans + a) % P;
    b >>= 1;
    a = a * 2 % P;
  }
  return ans;
}
```

- We are adding a, 2a, 4a, 8a, and so on if they exist in the binary representation of b.

We have Fibonacci numbers fib(0) = 0, fib(1) = 1, and fib(i) = fib(i-1) + fib(i-2) for i >= 2.

Compute fib(n) mod P = 10^9+7, n <= 10^9

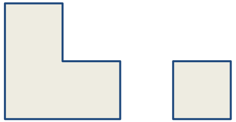
- Again, we cannot afford O(n=10^9) to linearly compute it.

$$M = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

$$M \cdot \begin{bmatrix} fib(i-1) \\ fib(i) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} fib(i-1) \\ fib(i) \end{bmatrix} = \begin{bmatrix} fib(i) \\ fib(i+1) \end{bmatrix}$$

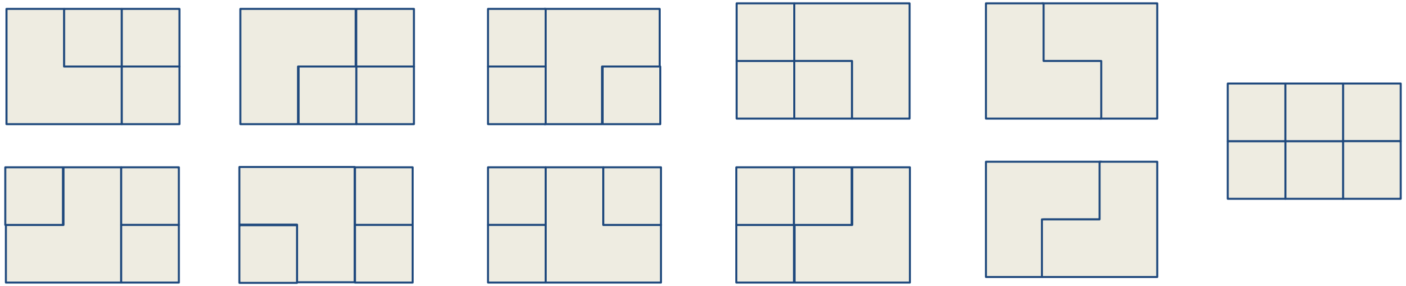
$$M^{n-1} \cdot \begin{bmatrix} fib(0) \\ fib(1) \end{bmatrix} = \begin{bmatrix} fib(n-1) \\ fib(n) \end{bmatrix}$$

We can compute M^{n-1} using logarithm power (with modulus). The time complexity is $O(\log n * k^3)$, where the matrix is of size k-by-k.



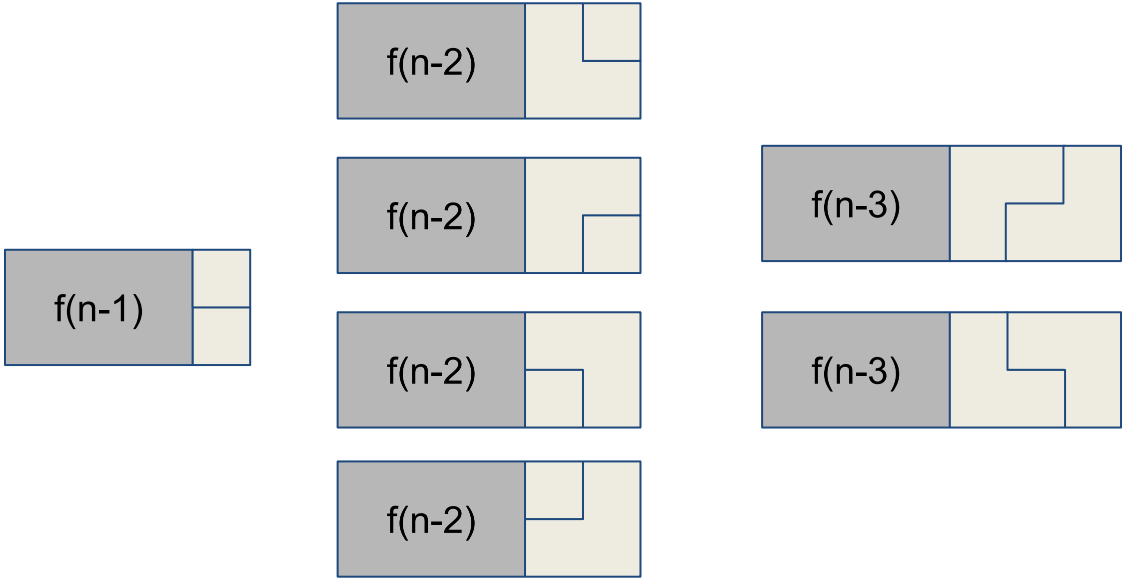
Count how many ways to use two types (L shape and single cell) of blocks to construct a wall of size 2 x N ($N \le 10^9$). L shape can be rotated.

Sample: 11 ways for N = 3



Blocks Recurrence

Let f(n) be the number of ways to build a wall of 2 x n.
We have f(n) = f(n-1) + 4f(n-2) + 2f(n-3), for n >= 3.
Boundary: f(0) = 1, f(1) = 1, f(2) = 5



f(n) = f(n-1) + 4f(n-2) + 2f(n-3)

$$M = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 2 & 4 & 1 \end{bmatrix} \qquad M \cdot \begin{bmatrix} f(n-3) \\ f(n-2) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} f(n-2) \\ f(n-1) \\ f(n) \end{bmatrix}$$

Time complexity: O(logn * 3^3)

- Form a monotonic search space.
- Check the middle point of the search space.
- Reduce the search space by half.

Element Existence

Given an array of N integers ($N<=10^5$), and Q queries. Each query has a number x and asks if x is in the array.

- Q = 1: Linear scan
- $Q = 10^5$: Sort the array before all the queries, and binary search for each query. Time: O(QlogN)
    - Search space is index range l = 0, r = N-1
    - Pick the middle index m = (l + r)/2, if arr[m] < x, then search space becomes [m + 1, r]; Otherwise, search space becomes [l, m - 1].

Binary Search Implementation

- There are binary search util functions for binary searching on a sorted array.
- However we must know how to write a binary search ourselves because for many problems the search space is not just an array.

```
int binarySearch(int l, int r) {
  while (l <= r) {
    int m = (l + r) / 2;
    if (condition(m)) r = m - 1;
    else r = m + 1;
  }
  return l; // or r
}
```

monotonic search space

`l   F F F F F F F F F F F F F F F F F T T T T T T T T T T T T T T T T T T T T   r`

`r l` Last positions of l and r when the loop ends

Want the smallest i so that condition(i) = true, pick `l`.
Want the largest i so that condition(i) = false, pick `r`.

- Be careful with binary search's search space reduction and exit condition.
    - Some people use while (l < r) in place of while (l <= r)
    - Some people use l = m, r = m when reducing the search space by half.
- The while loop may not terminate if this is not set correctly!

- If l and r are floating point numbers, then we can just loop for a constant number of times (e.g. 100 times).
    - Each time the range is cut into half, and after k times we have a precision of 2^{-k}

- Sometimes for a problem that asks for an answer X, directly computing X can be difficult.
- However, given X, checking if X is an answer is easy.

- If the problem's solution space is monotonic, then we can binary search the solution space.
  - Monotonic solution space example: If X works, then every Y > X also works.

Given a count k ($k<=10^9$), find the smallest integer n, so that n! has at least k trailing zeroes.

- Directly finding such a smallest n is difficult (or tedious if not impossible).
- However, given n calculating how many trailing zeroes n! has is straightforward.

NYU

Trailing Zeroes in a Factorial

- The number of trailing zeroes in n! only depends on how many factors of 2's and 5's n! has.
- For n! the number of 2's is always greater than the number of 5's, so we can simply count the number of 5's.

	5	10	15	20	25	30	...	50	…	125
5^1 (1st 5)	✓	✓	✓	✓	✓	✓		✓		✓
5^2 (2nd 5)					✓			✓		✓
5^3 (3rd 5)										✓

Therefore trailing zeroes in n! equals floor($n/5^1$) + floor($n/5^2$) + floor($n/5^3$) + ...

Solution

- Binary search on the answer n
- Initial range is [l, r] = [1, 10^9]
- Let m = (l + r)/2, count the number of trailing zeroes in m!, say t.
  - If t >= k: [l, r] ← [l, m - 1]
  - else: [l, r] ← [m + 1, r]
- Here, condition is t >= k, we want the *smallest* value where the condition holds. So finally we return the last value of l.

Time: O($\log^2$n). We binary search on n which is O(logn). Counting the number of trailing zeroes for a given m is also O(logn).