Algolab 2018 – Week 2

A couple of general remarks:

- **Public** test sets are different than the test sets on the **judge**
- In particular, judge test sets **may** (or may not) contain special edge cases not existing in the public ones
- Think about test cases for which your algorithm might **fail** or possibly behave strangely
- algolab@lists.inf.ethz.ch doesn't receive emails from non ETHZ accounts
- Fedora will be the exam OS
- We have updated the C++ introduction document and virtualbox image

Today's lecture

- **Three** different techniques:
  - Binary search
  - Sliding window
  - Dynamic programming

Binary search

Binary search

Problem: Barber shop

A barber takes L minutes to cut every customer's hair. How to pick L?

- n customers arrive at times $t_0, \ldots, t_{n-1}$.
- The last customer must leave exactly at time T.
- The barber starts cutting the next customer's hair as soon as he is finished with his current customer.
- But maybe he has to wait for the next customer to arrive.
- Find L

Binary search

Problem: Barber shop example

- 3 customers arrive at times $0, 2, 10$.
- The last customer must leave exactly at time 14.
- With $L = 1$
  - First customer arrives at 0 and leaves at 1
  - Second customer arrives at 2 and leaves at 3
  - Last customer arrives at 10 and leaves at 11
  - Last customer leaves before time 14 **:(**

Binary search

Problem: Barber shop example

- 3 customers arrive at times $0, 2, 10$.
- The last customer must leave exactly at time 14.
- With $L = 4$
  - First customer arrives at 0 and leaves at 4
  - Second customer arrives at 2 and leaves at 8
  - Last customer arrives at 10 and leaves at 14
  - Last customer leaves exactly at time 14 **:)**

Reformulation

Let us say that L is too small if the last customer will leave before time T.

We need to find the smallest L that is not too small.

Simpler problem: check whether a **fixed** L is too small.

To check if a fixed L is too small:

- Precomputation: sort the arrival times in increasing order.

```
sort(t.begin(), t.end());
```

- Check if L is too small:

```
bool too_small(int L) {
    int time = 0;
    for (int i = 0; i < n; i++) {
        if (t[i] <= time)
            time += L;
        else
            time = t[i] + L;
    }
    return (time < T);
}
```

Trick/technique (Sorting)

Sorting can be a powerful pre-computation step.

To sort a vector, always use the `std::sort` function from the `<algorithm>` library.

- $\mathcal{O}(n \log n)$ time for the pre-computation
- $\mathcal{O}(n)$ time per call of `too_small`

How can we find the **smallest** L that is not too small?

Binary search

- The property of not being too small is monotone.
- So we can use **binary search** to efficiently find the smallest L with this property.
- First, we find an upper bound on the smallest L (**exponential search**):

```
int lmin = 0, lmax = 1;
while (too_small(lmax)) lmax *= 2;
```

- Now we do the binary search:

```
while (lmin != lmax) {
    int p = (lmin + lmax)/2;
    if (too_small(p))
        lmin = p + 1;
    else
        lmax = p;
}
L = lmin;
```

- In general, `std::lower_bound` and `std::upper_bound` can be useful.

In general

The problem: find the smallest k that is 'large enough'.

For a **fixed** k, you can check efficiently if it is 'large enough'.

How to find the smallest k efficiently?

Trick/technique (Binary search)

In such situations, we can use **binary search** to find the optimal k.

The running time is multiplied only with a factor of $\mathcal{O}(\log K)$, where K is the smallest k that is large enough.

Binary search: Takeaways

- **Sorting** is often a useful preprocessing step.
- No need to **explicitly** construct the search space.
- Exponential search gives an **upper-bound** for applying binary search.
- Values can get very **large**, be careful with using **int**.

Sliding Window

Problem of the week: Deck of Cards (Simplified)

Given positive numbers $v_0, \ldots, v_{n-1}$, find $0 \leq i \leq j < n$ such that:

$$k = \sum_{\ell=i}^{j} v_\ell$$

Example: $k = 7$

3	1	4	1	1	8

Solution: $i = 1$ and $j = 4$

Problem of the week: Deck of Cards (Simplified)

Given positive numbers $v_0, \ldots, v_{n-1}$, find $0 \leq i \leq j < n$ such that:

$$k = \sum_{\ell=i}^{j} v_\ell$$

- Test case 1: $n < 200 \rightarrow\ O(n^3)$
- Test case 2: $n < 3000 \rightarrow\ O(n^2)$
- Test case 3 and 4: $n < 10^5 \rightarrow\ O(n)$

It is possible to solve this in time $\mathcal{O}(n)$:

Idea:

- Keep two pointers to keep track of current sequence
- If the current sequence is too **small**: Increase the right pointer
- If the current sequence is too **large**: Increase the left pointer

$k = 7$

3	1	4	1	1	8

It is possible to solve this in time $\mathcal{O}(n)$:

```
int i = 0, j = 0;
int sum = v[0];

while (1) {
    if (sum == k) break;

    if (sum < k) {
        j++;
        sum += v[j];
    }
    else {
        sum -= v[i];
        i++;
    }
}
```

Why this works (sketch):

3	1	4	1	1	8

- Assume left pointer reaches first the beginning of the optimal sequence.
- Right pointer will keep increasing until it reaches the end.
- Same argument if right pointer reaches first.

Trick/technique (Sliding window)

Some problems in which you need to find some **optimal interval** can be solved in linear time using a similar **sliding window** approach if you can ensure the optimal interval will be considered.

Dynamic Programming

Dynamic Programming – Outline

- Most of you know Dynamic Programming (DP) :)
- Many struggle to apply it :(
  - How to identify a DP problem
  - How to tackle it
  - How to implement it

First Example: Fibonacci Numbers

Definition: $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 2$

Task: compute F_n.

Solution: transform definition into recursive algorithm.

```
int f(int i) {
    if(i == 1 || i == 2) return 1;
    return f(i-1) + f(i-2);
}
```

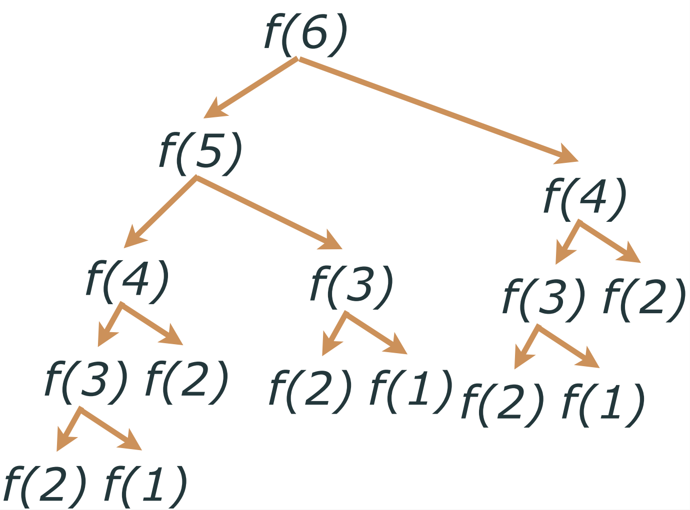
Time complexity: $\Theta(\phi^n)$

Source of inefficiency? Overlapping Subproblems...

First Example: Fibonacci Numbers

Definition: $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 2$

Source of inefficiency? Overlapping Subproblems...



Fibonacci Numbers – Memoization

Recall: $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 2$

Idea: do not recompute, recall from memory

```
vector<int> memo(n + 1, -1);

int f(int i) {
    if(i == 1 || i == 2) return 1;    // Check base cases
    if(memo[i] != -1) return memo[i]; // Check table
    int result = f(i-1) + f(i-2);     // Compute results
    memo[i] = result;                 // Store result to table
    return memo[i];
}
```

Time complexity: $\Theta(n)$

Memoization (or top-down DP) is simple and powerful :)

This was easy. Why is it difficult in general?

Essence of DP:

- Compute a solution from solutions of subproblems.
- Solve subproblems only once, by storing results.

Storing results is easy, just apply memoization.

Deriving a recursive algorithm is the difficult part!

Usually we do not get a recursive definition of the problem... :(

Silly implementation details can cost you **many** points

Common pitfalls

New Task: $F_{-1} = 1$, $F_0 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 0$

```
vector<int> memo(n + 1, -1);

int f(int i) {
    if(memo[i] != -1) return memo[i]; // Check table
    if(i == -1 || i == 0) return 1;   // Check base cases
    int result = f(i-1) + f(i-2);     // Compute results
    memo[i] = result;                 // Store result to table
    return memo[i];
}
```

This will get a **RUN-ERROR**, can you spot why?

Many times the **base cases** will not be in the memory table

Common pitfalls

New Task: Compute parity of Fibonacci number

```
vector<int> memo(n + 1, 0);

int f(int i) {
    if(i == 1 || i == 2) return 1;      // Check base cases
    if(memo[i] != 0) return memo[i]     // Check table
    int result = (f(i-1) + f(i-2)) % 2; // Compute results
    memo[i] = result;                   // Store result to table
    return memo[i];
}
```

This will get a **TIMELIMIT**, can you spot why?

Make sure the **default** memory value is **not** a possible output

Second Example: Drink as much as possible

Task: given a sequence of n bottles with volumes $v_1, \ldots, v_n$. Drink as much as you can, without drinking from two adjacent bottles.

Example: 3 bottles with volumes 1, 4, 2

- Drink from first and last bottle $\rightarrow 3$ litres
- Drink from second bottle $\rightarrow 4$ litres

Second Example: Drink as much as possible

Task: given a sequence of n bottles with volumes $v_1, \ldots, v_n$. Drink as much as you can, without drinking from two adjacent bottles.

We want a recursive definition for $f(i) :=$ "max amount we can drink from first i bottles".

- Base cases: $f(0) = 0$ and $f(1) = v_1$.
- $f(i) = \max\{v_i + f(i-2), f(i-1)\}$

Now we can transform this definition into a recursive algorithm.

```
int f(int i) {
if (i == 0) return 0;          // Check base cases
if (i == 1) return volumes[i]; // Check base cases
return max(volumes[i] + f(i-2), f(i-1));
}
```

Time complexity: $\Theta(\phi^n)$ (same as Fibonacci)

Drink as much as possible – Memoization

Take recursive algorithm

```
int f(int i) {
    if (i == 0) return 0;          // Check base cases
    if (i == 1) return volumes[i]; // Check base cases
    return max(volumes[i] + f(i-2), f(i-1));
}
```

and simply add memo:

```
vector<int> memo(n + 1, -1);

int f(int i) {
    if (i == 0) return 0;              // Check base cases
    if (i == 1) return volumes[i];     // Check base cases
    if(memo[i] != -1) return memo[i]; // Check table
    memo[i] = max(volumes[i] + f(i-2), f(i-1)); //Store results
    return memo[i];
}
```

Time complexity: $\Theta(n)$

General Strategy

Find recursive formulation for the problem.

Implement it as recursive algorithm, it will be correct but slow.

Are there overlapping subproblems?

Add memoization!

What is left?

We focus on examples to illustrate particular difficulties that often occur in problems.

- Iterative DP (table, bottom-up)
- Compare memoization and iterative DP
- Reconstruct solutions
- Example with “complicated and many” subproblems

Drink as much as possible – Iterative DP

Recall: $f(0) = 0$ and $f(1) = v_1$ and $f(i) = \max\{v_i + f(i-2), f(i-1)\}$

We can easily transform this into an iterative algorithm:

```
int f(int n) {
    vector<int> dp(n+1); // dp table
    dp[0] = 0;
    dp[1] = volumes[1];
    for(int i = 2; i <=n; ++i)
        dp[i] = max(volumes[i] + dp[i-2], dp[i-1]);
    return dp[n];
}
```

The DP table follows naturally from recursive definition.

Memoization vs Iterative DP

Usually both work, so use what feels more natural to you ;)

Memoization:

- Simple (once you have recurrence)
- Easy to use other subproblem descriptions (e.g. sets...) by using a map
- Only computes necessary subproblems
- Overhead of function calls
- Sometimes time complexity not obvious

Iterative DP (with table):

- More effort to code
- Need to describe subproblems with integers
- Computes always all subproblems
- Time complexity obvious

Drink as much as possible – Reconstruct Solution

We computed how much we can drink. What if we want to know which bottles to take?

1. Compute DP table or memo
2. Reconstruct solutions using recurrence and a stack that remembers where we come from.

Recall recurrence: $f(0) = 0$, $f(1) = v_1$, $f(i) = \max\{v_i + f(i-2), f(i-1)\}$

```
stack<int> partial; // partial solutions
void reconstruct(int i) {
    if(i == 0) return; // p contains a solution
    if(i == 1) {partial.push(1); return;} // p contains solution

    if(volume[i] + dp[i-2] > dp[i-1]){ // we took the i-th bottle
        partial.push(i);
        reconstruct(i-2);
    }else // we did not take the n-th bottle
        reconstruct(i-1);
    }
}
```

Last Example: Longest Increasing Subsequence in $\Theta(n^2)$

Task: given a sequence of n integers $a_1, \ldots, a_n$. Compute the length of a longest increasing subsequence (LIS).

First attempt: $f(i) :=$ "length of LIS in $a_1, \ldots, a_i$".

- Base cases: $f(0) = 0$
- $f(i) = ???$

Final attempt: $f(i) :=$ "length of LIS in $a_1, \ldots, a_i$ that ends in a_i".

- Base cases: $f(0) = 0$
- $f(i) = \max_{j<i : a_j \leq a_i} \{1 + f(j)\}$

Solution: $\max_i f(i)$

We had to reformulate the problem s.t. it admits a recursive formulation, this is difficult!

Time complexity: n function calls (with memo), i-th call takes $\Theta(i)$ time. Thus, $\Theta(n^2)$.

DP – Wrap Up

- Idea of DP: solve subproblems only once by storing solutions of subproblems
- Start by defining recurrence relation (on paper)
- Implement it. It will be correct but slow...
- Are there overlapping subproblems?
- Add memo (usually this does the trick) or construct DP table
- Practice finding recurrence relation on paper for well known DP problems (SubsetSum, Knapsack, Coin Change, LCS, Edit Distance, LIS...)

That's all for today!