Technical Interviews

Kenny Yu

Handouts

- Resources
  - Contains some general tips and strategies for an interview. Also contains references to books and online resources.
- Problems
  - List of problems I've compiled.
  - DISCLAIMER: Your interview preparation *should not be limited to this list*. It is simply to give you a flavor of the kinds of problems you might see in an interview.
- All handouts, code, and presentation can be found here:
  - https://github.com/kennyyu/workshop

Structure of the Interview

- Typically 30-45 minutes, multiple rounds depending on the company
- Coding on the whiteboard if onsite, otherwise coding in a Google Doc or collabedit over the phone
- Typically 2-3 problems testing your computer science knowledge, will probably involve coding

Types of Questions

- Problem Solving
  - Find optimal solutions (big O) in terms of time and space, then code it up.
- Data Structures & Algorithms
  - Know when and how to use a data structure and/or algorithm
- Designing
  - Design the interfaces/classes/modules of a system. How do the components of the system interact?
  - Design patterns (Gang of Four book)

General Tips before Diving In

1. Always be thinking out loud. The point of the interview is to gauge how you think through a problem. If you are silent, the interviewer will learn nothing about your thinking process.

2. Repeat the question back in your own words.

3. Make sure you understand the problem by working through a few small and simple test cases. This will give you time to think and get some intuition on the problem. Your test cases should cover all normal and boundary cases (null, negatives, fractions, zero, empty, etc.).

4. Write down the function header/interface/class definition first and validate it with your interviewer to make sure you understand the problem.

5. Don't get frustrated.

General Tips before Diving In

6. Don't try to come up with the most efficient algorithm from the first go. Propose the simplest (slow, but correct) algorithm you can think of and then start thinking of better solutions. This could mean brute forcing the problem (trying all cases). Point out the inefficiencies of this solution (e.g. time and/or space complexity). This will also give you a starting point from which to find a more efficient solution.

7. Code design. What helper classes do I need (e.g. Points, Pair, Dates, ...)? What helper functions/methods do I need?

8. When you are done writing your code, validate your code on your test cases.

Writing a shuffle

Given an array of n integers, write a function that shuffles the array in place such that all permutations of the n integers are equally likely.

Assume you have available a random integer generator to generate a random integer in the range [0,i) for any positive integer i.

Ideas?

```
void shuffle(int array[], int n);

O(n^2) ideas?

O(n log n) ideas?

O(n) ideas?
```

Knuth Shuffle

- Also known as Fisher-Yates shuffle.
- O(n) shuffle, where n is the length of the array!
- Start with i = 0. Pick a random location j in the un-shuffled part of the array [i,n-1]. Swap array[i] and array[j]. Increment i and repeat until i == n – 1.

Knuth Shuffle

```
/** Swaps the values at addresses pointed to by x and y. */
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

/** Shuffles the array of length @a n such that
 * all permutations are equally likely. */
void shuffle(int *array, int n) {
  for (int i = 0; i < n; i++) {
    swap(&array[i], &array[my_random(i, n)]);
  }
}
```

Knuth Shuffle

- Can you ***prove*** the correctness (each permutation equally likely) of the algorithm?

Knuth Shuffle

- Can you ***prove*** the correctness (each permutation equally likely) of the algorithm?
    - Induction! Try it!

Maximal Subarray

- Given an array of integers (positive/zero/negative), write a function that calculates the maximum sum of any continuous subarray of the input array.
- Example:
  - [1,2,3,4] -> 1 + 2 + 3 + 4 = 10
  - [2,3,-1,-3] -> 2 + 3 = 5
  - [-1,5,100,-1000] -> 5 + 100 = 105
  - [-1,-2,-3,-4] -> 0
  - [1000,20000,-1,300000000] -> whole array

Ideas?

```
int maximal_subarray(int array[], int
   n);
```

```
O(n^2) ideas?

O(n log n) ideas?

O(n) ideas?
```

Stupid Idea first: Brute force

- Take all possible (start,end) pairs and compute the maximum sum
- Big O?

Stupid Idea first: Brute force

```
/** Brute force approach */
int maximal_subarray_dumb(int *array, int n) {
  int max_sum = 0;
  for (int start = 0; start < n; start++) {
    for (int end = start; end < n; end++) {
      int sum = 0;
      for (int i = start; i <= end; i++) {
        sum += array[i];
      }
      max_sum = MAX(max_sum, sum);
    }
  }
  return max_sum;
}
```

Better Idea: Dynamic Programming!

- This problem uses an idea called *dynamic programming*, which you will become very familiar with if you take CS 124.
- Idea: build up solutions to smaller problems first.

Better Idea: Dynamic Programming!

- This problem uses an idea called *dynamic programming*, which you will become very familiar with if you take CS 124.
- Idea: build up solutions to smaller problems first.
  - We currently have to worry about two things: where to start and where to end.

Better Idea: Dynamic Programming!

- This problem uses an idea called *dynamic programming*, which you will become very familiar with if you take CS 124.
- Idea: build up solutions to smaller problems first.
  - We currently have to worry about two things: where to start and where to end.
  - Wouldn't it be great if we can get rid of one of these parameters?

Dynamic Programming

- Original Problem: Find the maximum sum of any subarray in the range [0,n-1]
- Subproblem: Find the maximum sum of any subarray starting at any index in the range [0,i] and ending at i.

Dynamic Programming

- Original Problem: Find the maximum sum of any subarray in the range [0,n-1]
- Subproblem: Find the maximum sum of any subarray starting at any index in the range [0,i] and ending at i.
- What is my recurrence?

Dynamic Programming

- Original Problem: Find the maximum sum of any subarray in the range [0,n-1]
- Subproblem: Find the maximum sum of any subarray starting at any index in the range [0,i] and ending at i.
- What is my recurrence?
    - subproblem(i) = max(subproblem(i-1) + array[i], 0)

Dynamic Programming

- Original Problem: Find the maximum sum of any subarray in the range [0,n-1]
- Subproblem: Find the maximum sum of any subarray starting at any index in the range [0,i] and ending at i.
- What is my recurrence?
  - subproblem(i) = max(subproblem(i-1) + array[i], 0)
- Once we have built up this table of solutions, we can do a linear search across the table to find the largest element.

Dynamic Programming

```
int maximal_subarray_space(int *array, int n) {
  // build out solution array
  int *subproblems = (int *) malloc(n * sizeof(int));
  subproblems[0] = MAX(array[0], 0);
  for (int i = 1; i < n; i++) {
    subproblems[i] = MAX(subproblems[i-1] + array[i], 0);
  }

  // iterate through to find the maximum
  int max = 0;
  for (int i = 0; i < n; i++) {
    max = MAX(max, subproblems[i]);
  }
  free(subproblems);
  return max;
}
```

Dynamic Programming

```
int maximal_subarray_space(int *array, int n) {
  // build out solution array
  int *subproblems = (int *) malloc(n * sizeof(int));
  subproblems[0] = MAX(array[0], 0);
  for (int i = 1; i < n; i++) {
    subproblems[i] = MAX(subproblems[i-1] + array[i], 0);
  }

  // iterate through to find the maximum
  int max = 0;
  for (int i = 0; i < n; i++) {
    max = MAX(max, subproblems[i]);
  }
  free(subproblems);
  return max;
}
```

How much space does this algorithm use?

Dynamic Programming

```
int maximal_subarray_space(int *array, int n) {
  // build out solution array
  int *subproblems = (int *) malloc(n * sizeof(int));
  subproblems[0] = MAX(array[0], 0);
  for (int i = 1; i < n; i++) {
    subproblems[i] = MAX(subproblems[i-1] + array[i], 0);
  }

  // iterate through to find the maximum
  int max = 0;
  for (int i = 0; i < n; i++) {
    max = MAX(max, subproblems[i]);
  }
  free(subproblems);
  return max;
}
```

How much space does this algorithm use?

O(n)

Can we do better?

Dynamic Programming

```
int maximal_subarray_space(int *array, int n) {
  // build out solution array
  int *subproblems = (int *) malloc(n * sizeof(int));
  subproblems[0] = MAX(array[0], 0);
  for (int i = 1; i < n; i++) {
    subproblems[i] = MAX(subproblems[i-1] + array[i], 0);
  }

  // iterate through to find the maximum
  int max = 0;
  for (int i = 0; i < n; i++) {
    max = MAX(max, subproblems[i]);
  }
  free(subproblems);
  return max;
}
```

Note that we only care about the previous subproblem, and maximum ever we've seen so far…

Dynamic Programming

```
/** Dynamic programming approach. */
int maximal_subarray(int *array, int n) {
  int max_sum_so_far = 0;
  int current_sum_so_far = 0;
  for (int i = 0; i < n; i++) {
    current_sum_so_far = MAX(current_sum_so_far + array[i], 0);
    max_sum_so_far = MAX(max_sum_so_far, current_sum_so_far);
  }
  return max_sum_so_far;
}
```

Dynamic Programming

```
/** Dynamic programming approach. */
int maximal_subarray(int *array, int n) {
  int max_sum_so_far = 0;
  int current_sum_so_far = 0;
  for (int i = 0; i < n; i++) {
    current_sum_so_far = MAX(current_sum_so_far + array[i], 0);
    max_sum_so_far = MAX(max_sum_so_far, current_sum_so_far);
  }
  return max_sum_so_far;
}
```

Constant space!

Dynamic Programming

- Gives us an O(n) solution with O(1) space. Nice.
- Can we still do better?

Dynamic Programming

- Gives us an O(n) solution with O(1) space. Nice.
- Can we still do better?
  - No!
    - Need to at least read the input to the problem, so at least O(n) time.
    - Can’t do better than constant space!

Stock Market Problem

- Given an array of n integers (positive/zero/negative) that represent the price of a stock over n days, write a function to compute the maximum profit you can make, given that you buy and sell exactly 1 stock within these n days.

Ideas?

```
int stocks(int array[], int n);
```

```
O(n^2) solutions?

O(n log n) solutions?

O(n) solutions?
```

One idea: Divide & Conquer

- Use recursion (think merge sort)

One idea: Divide & Conquer

- Use recursion (think merge sort)
- Compute the best buy-sell pair in left half
- Compute the best buy-sell pair in right half
- How do we "merge" these solutions back together?

One idea: Divide & Conquer

- Use recursion (think merge sort)
- Compute the best buy-sell pair in left half
- Compute the best buy-sell pair in right half
- How do we "merge" these solutions back together?
- Compute the best buy-sell pair where we buy in left half and sell in right half.
    - Compute the lowest price in first half, compute the highest price in right half, and then take the difference
- Compute the maximum of these three

Divide & Conquer

```
int stocks_dc(int *array, int n) {
  if (n == 0) {
    return 0;
  }
  if (n == 1) {
    return MAX(array[0], 0);
  }
  int half = n / 2;
  int left_stocks = stocks_dc(array, half);
  int right_stocks = stocks_dc(array + half, n - half);
  int left_min = array[0];
  for (int i = 0; i < half; i++) {
    left_min = MIN(left_min, array[i]);
  }
  int right_max = array[half];
  for (int i = half; i < n; i++) {
    right_max = MAX(right_max, array[i]);
  }
  int mid_stocks = right_max - left_min;
  return MAX(left_stocks, MAX(right_stocks, mid_stocks));
}
```

Divide & Conquer

- What’s the Big O?

Divide & Conquer

- What's the Big O?
- Let T(n) = # steps to find best stock profit with n days

Divide & Conquer

- What's the Big O?
- Let T(n) = # steps to find best stock profit with n days
    - $T(n) = 2\,T(n/2) + c * n$
    - Time to find answer in left half, then right half, then O(n) work to merge solutions together (find the min in left half, find max in right half)

Divide & Conquer

- What's the Big O?
- Let T(n) = # steps to find best stock profit with n days
  - $T(n) = 2\,T(n/2) + c * n$
  - Time to find answer in left half, then right half, then O(n) work to merge solutions together (find the min in left half, find max in right half)
  - This is the recurrence equation for merge sort!!
  - So $O(n \log n)$

Divide & Conquer

- What's the Big O?
- Let T(n) = # steps to find best stock profit with n days
  - T(n) = 2 T(n/2) + c * n
  - Time to find answer in left half, then right half, then O(n) work to merge solutions together (find the min in left half, find max in right half)
  - This is the recurrence equation for merge sort!!
  - So O(n log n)
- How much space are we using?

Divide & Conquer

- What's the Big O?
- Let T(n) = # steps to find best stock profit with n days
  - T(n) = 2 T(n/2) + c * n
  - Time to find answer in left half, then right half, then O(n) work to merge solutions together (find the min in left half, find max in right half)
  - This is the recurrence equation for merge sort!!
  - So O(n log n)
- How much space are we using?
  - Our function isn't *tail recursive* (take CS51 to find out what this is), so therefore, the amount of space we use is the maximum number of stack frames we ever have at some moment

Divide & Conquer

- What's the Big O?
- Let T(n) = # steps to find best stock profit with n days
  - T(n) = 2 T(n/2) + c * n
  - Time to find answer in left half, then right half, then O(n) work to merge solutions together (find the min in left half, find max in right half)
  - This is the recurrence equation for merge sort!!
  - So O(n log n)
- How much space are we using?
  - Our function isn't *tail recursive* (take CS51 to find out what this is), so therefore, the amount of space we use is the maximum number of stack frames we ever have at some moment: O(log n) space

Ideas?

```
int stocks(int array[], int n);

O(n^2) solutions?

O(n log n) solutions?

O(n) solutions?
```

Can we do better?

Ideas?

```
int stocks(int array[], int n);
```

`O(n^2) solutions?`

`O(n log n) solutions?`

`O(n) solutions?`

Can we do better?

Can we reduce this to a problem we have already solved?

Reduction

- We can convert the stock market problem into the maximal subarray problem!
- How?

Reduction

- We can convert the stock market problem into the maximal subarray problem!
- How?
  - Think of the array in the maximal subarray as the *deltas* between the stock prices on consecutive days.
- To solve the stock problem, create a new array deltas[n] where deltas[0] = 0, and deltas[i] = stocks[i] – stocks[i-1]
  - Then apply the maximal subarray problem on the array deltas!

Reduction

```
int stocks(int *array, int n) {
  int *deltas = (int *) malloc(n * sizeof(int));
  deltas[0] = 0;
  for (int i = 1; i < n; i++) {
    deltas[i] = array[i] - array[i-1];
  }
  int best = maximal_subarray(deltas, n);
  free(deltas);
  return best;
}
```

Reduction

- $O(n)$ solution!
- Always a good idea: Try to reduce a new problem you encounter to a problem you have already solved.

Wrap-up

1. Technical interviews are challenging, but they can be very fun!

2. Practice, practice, practice.

3. Good luck on your interviews!

See handouts for more information:
https://github.com/kennyyu/workshop

If you are interested in a mock technical interview, email
Willie Yao, ‘13: wyao13@college.harvard.edu