How to Panic in a Coding Interview

Benjamin Toll
benjamintoll.com
benjam72@yahoo.com

Topics

- Interview Prep
- In the Interview
- Presentation Format
- Presentation Goals
- Time Complexity
- Space Complexity
- Examples of Common Runtimes
- Algorithms

  - Strings and Arrays

  - Linked Lists

  - Stacks and Queues

  - Trees

  - Bit Manipulation
- References

Interview Prep

- Practice in a language in which you’re not fluent (preferably one that doesn’t natively support all data structures).
- Read at least one book and use several websites.
- Study other people’s solutions.
- Practice all the time in blocks of time.
- Write the algorithms down using pencil and paper and only then type it into the computer.
- Want to at least get to the point where you think “ah, I’ve seen this one before”.

In the Interview

- Remember, the interview is a conversation. You want to know as much about them as they do about you.
- Don’t assume anything. Ask questions if you’re unsure about what has been asked.
- Demonstrate your knowledge:
    - If using a hash, talk about its how accessor functions are O(1) except in the case where there are collisions, which depends on the distribution of the hash function.
    - If using an array, talk about how insertions are constant time except when it occurs during a power of two, at which point the language could do table doubling. However, this cost is amortized since most insertions won’t trigger the table doubling.
- Draw it out!
- If stumped, try to at least do a brute-force solution.
- Meanwhile, think of a better runtime complexity and space complexity.
- Check for edge cases and trouble spots, such as off-by-one errors.
- Take a deep breath!

Presentation Format

- Question
- Interviewer imposes constraints
- Your analysis (talking out loud)

Presentation Goals

- We’re looking at the algorithms to get a feel for their complexity, both time and space.
- We’re not looking at Big O from an academic point of view, we just want to get more familiar and comfortable with talking about runtime complexity.
- As such, we’re not going to be analyzing the algorithms in detail, instead looking at the big picture.
- We’re not interested in doing a code review and analyzing every line of code.
- I chose examples that are interesting to me and/or have been asked in interviews.

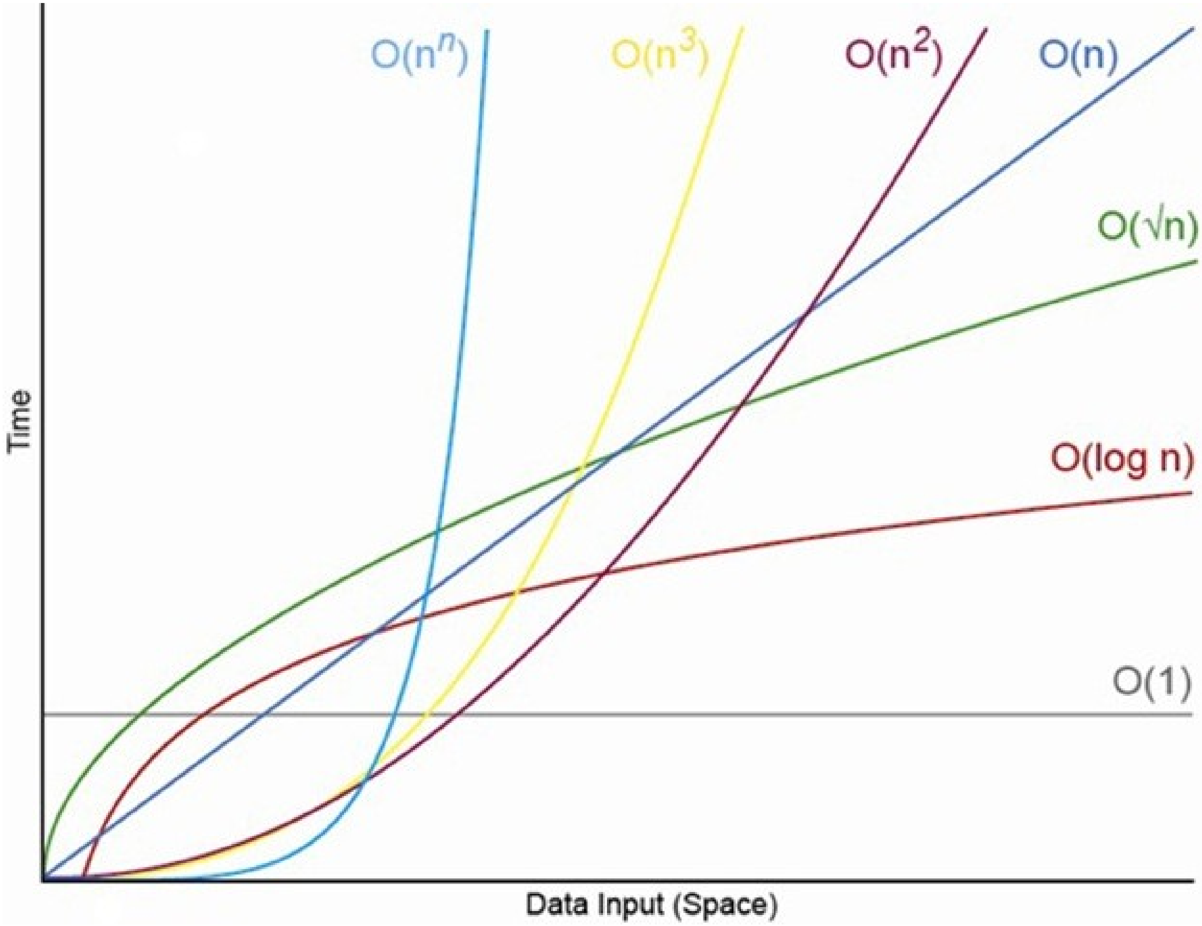
Time Complexity

- Also known as asymptotic runtime or Big O
- Gives us a language with which to measure the efficiency of an algorithm
- Common runtimes:
  - O(1) - constant
  - O(lg n) - logarithmic
  - O(n) - linear
  - O(n lg n) - linearithmic
  - O(n^2) - quadratic
  - O(2^n) – exponential
  - O(n!) - factorial
- Asymptotic notations
  - Big O – upper bound on the runtime (i.e., “it’s at least as fast as O(n), O(n lg n), O(n^2), etc.”)
  - Big Theta Θ – tight bound on the runtime, means both Big O and Big Omega
  - Big Omega Ω – lower bound on the runtime (i.e., “it’s at least as slow as O(n), O(lg n)...O(1)”)
- When programmers speak of Big O, they usually mean Big Theta
- For each runtime, there are best, worst and expected cases
- Only concerned with the higher-order terms, everything else is dropped

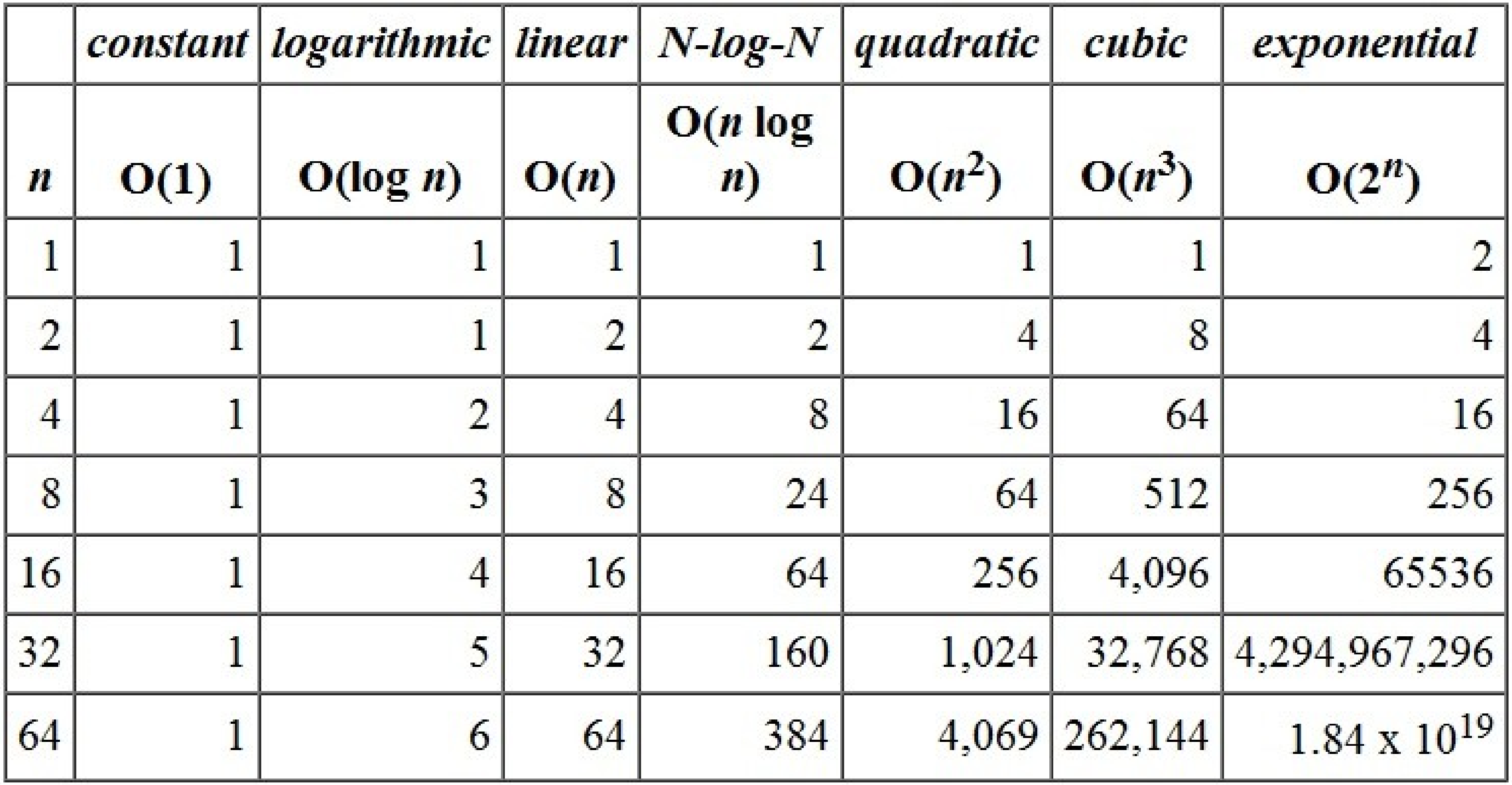
Space Complexity

- The amount of memory required by an algorithm
- Parallel concept to time complexity:
  - An array of size `n` requires O(n) space
  - A matrix (2d array) of size `n` x `n` requires O(n^2) space

Common Runtimes



Runtime Comparisons



	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84 x 10^{19}

Examples of Common Runtimes

```
void rot3(char* s) {
    int k = 0;

    while (*(s + k)) {
        char c = *(s + k);

        if (c >= 'a' && c <= 'z')
            *(s + k) = ((c + 3 - 'a') % 26) + 'a';

        k++;
    }
}

void main(int argc, char** argv) {
    if (argc == 1) {
        printf("Usage: %s <string>\n", argv[0]);
        exit(1);
    }

    char *s = argv[1];
    rot3(s);
    printf("%s\n", s);
}
```

$O(n)$

```
./rot3 "pq!rst uvwxyz_abc"
```

```
size_t len(char* s) {
    size_t k = 0;
    while (*(s++)) k++;
    return k;
}

void rot3(char* s) {
    for (int i = 0; i < len(s); i++) {
        char c = s[i];

        if (c >= 'a' && c <= 'z')
            s[i] = ((c + 3 - 'a') % 26) + 'a';
    }
}

void main(int argc, char **argv) {
    char s[] = "pq!rst uvwxyz_abc";

    rot3(s);
    int k = 0;

    while (*(s + k)) {
        printf("%c", *(s + k++));
    }
}
```

O(n)

```
void main(int argc, char **argv) {
    for (int i = 25; i > -1; --i)
        printf("%c\n", 'a' + i);
}
```

O(1)

```
void main(int argc, char **argv) {
    if (argc < 2) {
        printf("Usage: %s <n>\n", argv[0]);
        exit(1);
    }

    int n = atoi(argv[1]);
    int matrix[n][n];
    int k = 0;

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            matrix[i][j] = k++;
}
```

$O(n^2)$

```
#include <stdio.h>

void main(int argc, char **argv) {
   char *s = argv[1];
   char *t = argv[2];

   while (*s)
      printf("%c\n", *s++);

   while (*t)
      printf("%c\n", *t++);
}
```

$O(a+b)$

```
int binary_search(int nodes[], int p, int r, int k) {
    if (p > r)
        return -1;

    int q = p + r >> 1;

    if (nodes[q] == k)
        return q;

    if (nodes[q] > k)
        binary_search(nodes, p, q - 1, k);
    else
        binary_search(nodes, q + 1, r, k);
}
```

O(lg n)

Strings and Arrays

Q. Determine if a string contains only unique characters

- Interviewer:
  - No constraints

Analysis

- What data structures could be used?
- If the constraints are open-ended, ask if you can do an operation to simplify the problem (like sorting).
- Whenever you can sort something, do it! The tradeoff is that the order of growth may increase, but it can greatly simplify the problem.
- For example, the solution may then be able to be done in constant space instead of using another data structure.

```
int main(int argc, char **argv) {
    char s[] = "yellowjacket";
    int len = strlen(s);

    if (len > 26) return 0;

    // sort string

    for (int i = 0; i < len; i++)
        if (s[i] == s[i + 1]) return 0;

    return 1;
}
```

O(n lg n)

Q. Determine if a string contains only unique characters (continued)

- Interviewer:
  - Do it in constant space

Analysis

- Can it be done in only one pass?
- What about a bit vector?
- Weeeeeeeeeeeeeeeeeeeeee

```
int main(int argc, char **argv) {
    char* s = "yellowjacket";
    int v = 0, k = 0;

    while (s[k]) {
        int shift = 1 << s[k++] - 'a';
        if ((v & shift) > 0) return 0;
        v |= shift;
    }

    return 1;
}
```

O(n)

Q. Remove duplicate characters in a string

- Interviewer:
  - No constraints

Analysis

- I know I can use another data structure, but...
- Can it be done using the same string?

```
void main(int argc, char **argv) {
    char s[] = "wrhelodldl";
    size_t l = sizeof(s) / sizeof(char);

    // mergesort

    int k = 0;
    for (int i = 0; i < l; ++i)
        if (s[i] != s[i + 1])
            s[k++] = s[i];

    s[k] = '\0';
    printf("%s\n", s);
}
```

$O(n \lg n)$

```
void main(int argc, char **argv) {
    char s[] = "hwwrhelodldl";
    int v = 0, k = 0;

    printf("%s\n", s);

    for (int i = 0; i < strlen(s); ++i) {
        int shifted = 1 << s[i] - 'a';

        if ((v & shifted) == 0) {
            v |= shifted;
            s[k++] = s[i];
        }
    }

    s[k] = '\0';
    printf("%s\n", s);
}
```

O(n)

Sometimes, the solution requires special domain knowledge.

This is another reason to always be practicing.

Q. Find missing integer in a sorted array

- Interviewer:
  - Integers are ordered in an arithmetic series
  - Do not use any extra data structures

Analysis

- The key to solving this is in the hint that the array is an arithmetic sequence, i.e., the difference between the numbers is constant.
- Developers with a math background will know that Gauss developed an elegant (and simple) method to find the aggregate of an arithmetic progression of 1s:
    - $(n^2 + n) / 2$
    - $n(n + 1) / 2$
    - $n = max$

```
void main(int argc, char **argv) {
    int n = 100, k = 0;
    int nums[n];

    for (int i = 0; i < n; ++i)
        nums[i] = i + 1;

    // "Randomly" reset one of the elements to 0.
    nums[53] = 0;

    for (int i = 0; i < n; ++i)
        k += nums[i];

    int gauss = (n * (n + 1)) / 2;

    printf("%d\n", gauss - k);
}
```

Q. Find unique integer in an unsorted array

- Interviewer:
  - All other integers in the array have a duplicate
  - Do not use any extra data structures

Analysis

- It would be simple to solve this using an array or hash, but this must be solved in constant space.
- What about a bitwise operation?
- XORing a number by itself is zero.

```
void main(int argc, char **argv) {
    int nums[] = { 3, 5, 7, 6, 8, 9, 7, 42, 6, 8, 5, 9, 3 };
    int k = 0;

    for (int i = 0; i < sizeof(nums) / sizeof(int); ++i)
        k ^= nums[i];

    printf("%d\n", k);
}
```

Linked Lists

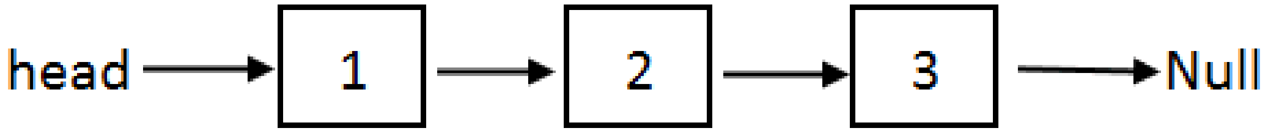
Q. Create a linked list

- Singly? Doubly? Ask!
- Should there be a reference to the tail?
- What about the API?

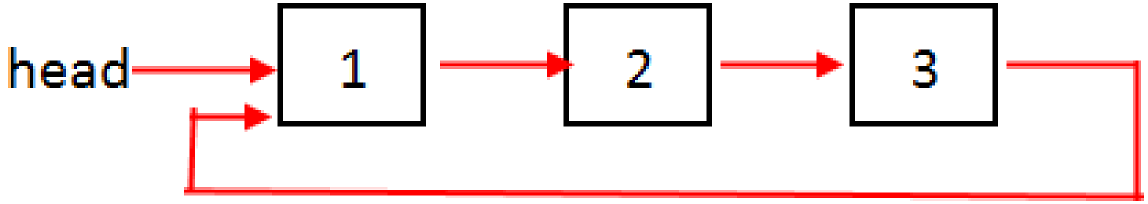
```
struct node {
    struct node *next;
    int val;
};

void add(struct node **head, int v) {
    struct node *n;

    if (!(n = malloc(sizeof(struct node)))) {
        fprintf(stderr, "Could not allocate memory for new node!");
        exit(1);
    }

    if (*head == NULL) {
        n->val = v;
        n->next = NULL;
        *head = n;
    } else {
        struct node *c = *head;

        while (c->next) {
            c = c->next;
        }

        n->val = v;
        n->next = NULL;
        c->next = n;
    }
}
```

```
void list(struct node **head) {
    struct node *n = *head;

    while (n) {
        printf("%d\n", n->val);
        n = n->next;
    }
}

struct node* find(struct node **head, int v) {
    if (*head == NULL) return NULL;
    if ((*head)->val == v) return *head;

    struct node *n = (*head)->next;

    while (n) {
        if (n->val == v) return n;
        n = n->next;
    }

    return NULL;
}
```

Q. Determine if a linked list is circular



Singly Linked List



Circular Linked List

```
#include "../linked_list.c"

int is_circular(struct node **node) {
    struct node *tortoise = *node;
    struct node *hare = *node;
    int i = 0;

    while (i++ < 2 && hare->next)
        hare = hare->next;

    while (tortoise->next && hare->next) {
        tortoise = tortoise->next;
        hare = hare->next;

        if (tortoise == hare) return 1;
        if (hare->next) hare = hare->next;
    }

    return 0;
}
```

```
void main(int argc, char **argv) {
    struct node *HEAD = NULL;

    for (int i = 1; i < 10; ++i)
        add(&HEAD, i * 2);

    list(&HEAD);

    struct node *tail = find(&HEAD, 18);
    tail->next = HEAD;

    printf("%d\n", is_circular(&HEAD));
}
```

Q. Delete a node in a linked list given only the node itself

- You are not given a reference to HEAD
- The linked list is not doubly-linked

```
#include "../linked_list.c"

void main(int argc, char **argv) {
    struct node *HEAD = NULL;

    for (int i = 1; i < 10; ++i)
        add(&HEAD, i * 2);

    struct node *to_delete = find(&HEAD, 10);

    if (to_delete->next) {
        to_delete->val = to_delete->next->val;
        to_delete->next = to_delete->next->next;
    } else {
        fprintf(stderr, "Can't delete tail!");
        exit(1);
    }
}
```

Q. Return the kth to last node

```
#include "../linked_list.c"

int kth_to_last(struct node **node, int k) {
    struct node *tortoise = *node;
    struct node *hare = (*node)->next;

    int j = 1;

    while (j++ < k && hare->next)
        hare = hare->next;

    if (j - 1 != k) {
        fprintf(stderr, "Error: list length is less than k");
        exit(1);
    }

    while (hare->next) {
        tortoise = tortoise->next;
        hare = hare->next;
    }

    return (*tortoise).val;     // Same as `tortoise->val`.
}
```

```
void main(int argc, char **argv) {
    struct node *HEAD = NULL;

    for (int i = 1; i < 10; ++i)
        add(&HEAD, i * 2);

    list(&HEAD);
    printf("\n%d\n", kth_to_last(&HEAD, 3));
}
```

Stacks and Queues

Q. Create a stack

- Implement

  - pop

  - push

  - size

```
#define MAX_N    100

struct stack {
    int data[MAX_N];
    int sp;     // stack pointer
};

struct stack* create_stack() {
    struct stack *s;

    if (!(s = malloc(sizeof(struct stack)))) {
        fprintf(stderr, "Error: Could not initialize stack");
        exit(1);
    }

    return s;
}
```

```
int pop(struct stack *s) {
    if (s->sp > 0)
        return s->data[--s->sp];
    else
        fprintf(stderr, "Error: Stack empty");
}

void push(struct stack *s, int v) {
    if (s->sp < MAX_N)
        s->data[s->sp++] = v;
    else
        fprintf(stderr, "Error: Stack full");
}

int size(struct stack *s) {
    return s->sp;
}
```

Q. Create a queue from two stacks

- Interviewer:
    - Implement
        - enqueue
        - dequeue
        - is_empty

```c
#include "../stack.c"

struct queue {
    struct stack *s1;
    struct stack *s2;
};

void enqueue(struct queue *q, int v) {
    if (size(q->s2) > 0)
        while (size(q->s2))
            push(q->s1, pop(q->s2));

    push(q->s1, v);
}

int dequeue(struct queue *q) {
    if (size(q->s1))
        while (size(q->s1))
            push(q->s2, pop(q->s1));

    return pop(q->s2);
}

int is_empty(struct queue *q) {
    return (size(q->s1) + size(q->s2)) > 0;
}
```

```c
void main(int argc, char **argv) {
    struct queue *q;

    if (!(q = malloc(sizeof(struct queue)))) {
        fprintf(stderr, "Error: Could not initialize queue");
        exit(1);
    }

    q->s1 = create_stack();
    q->s2 = create_stack();

    enqueue(q, 5);
    enqueue(q, 7);
    enqueue(q, 9);
    enqueue(q, 11);
    printf("%d\n", dequeue(q));
    enqueue(q, 3);

    printf("size s1 -> %d\n", size(q->s1));
    printf("size s2 -> %d\n", size(q->s2));
}
```

Q. Determine if parentheses are balanced

- Interviewer:
  - No constraints

```
#include "../stack.c"

void main(int argc, char **argv) {
    char *t = "He(ll(o) W)o()r(ld)";
    struct stack *s = create_stack();

    while (*t) {
        if (*t == '(') push(s, *t);

        if (*t == ')')
            if (size(s) == 0) push(s, *t);
            else pop(s);

        *t++;
    }

    printf("%d\n",
        size(s) == 0 ? 1 : 0);
}
```

Trees

Q. Determine if binary tree is a binary search tree

- Interviewer:
  - Remember that a node in a binary tree has 0, 1 or 2 children
  - Placement of a child node is not constrained by value

Analysis

- What are the non-variants of a binary search tree?
- What type of tree traversal is needed?
- Should the traversal be recursive or iterative?

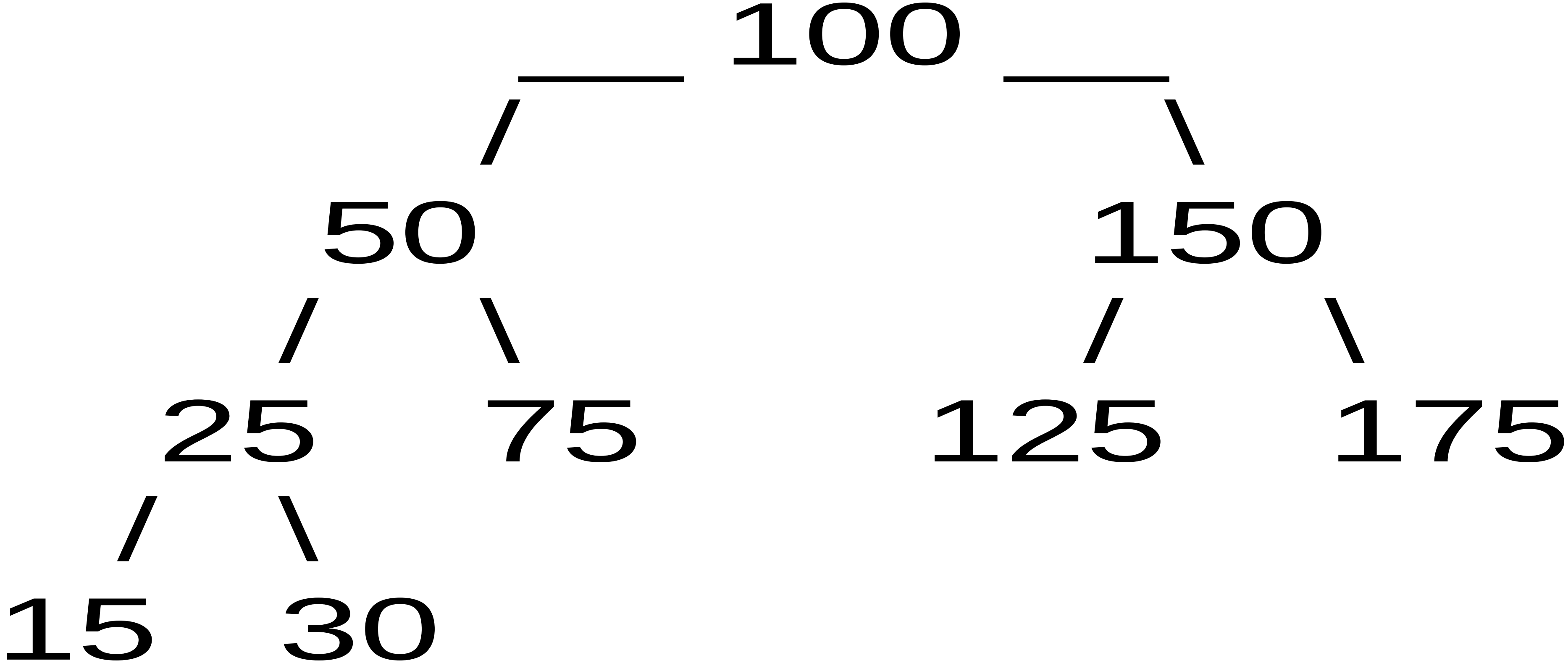
```
struct tree {
    struct tree_node *root;
    size_t count;
};
```

```
void main(int argc, char **argv) {
    struct tree *t;

    if ((t = malloc(sizeof(struct tree))) == NULL) {
        fprintf(stderr, "Could not allocate memory for new tree!");
        exit(1);
    }

    make_tree(t);

    int collector[t->count];
    int k = 0;

    traverse(t->root, &k, collector);

    for (int i = 1; i < t->count; ++i)
        if (collector[i] < collector[i - 1]) {
            printf("0\n");
            return;
        }

    printf("1\n");
}
```

```
void traverse(struct tree_node *n, int *k, int nodes[]) {
    if (n == NULL) return;
    traverse(n->left, k, nodes);
    nodes[*k] = n->val;
    *k = *k + 1;
    traverse(n->right, k, nodes);
}
```

Q. Serialize a binary search tree

- Interviewer:
  - Do it iteratively



```
            ___ 100 ___
           /           \
         50             150
        /  \           /   \
      25    75       125    175
     /  \
   15    30
```

Bit Manipulation

Q. Determine if n is a power of two

Analysis

- What does it mean for a number to be a power of two?
- Is there a pattern?
- Could we look at each bit in turn?

```
int is_power_of_two(int v) {
    if (v == 1) return 1;

    do {
        if (v & 1 != 0) return 0;
    } while ((v >>= 1) > 1);

    return 1;
}

void main(int argc, char **argv) {
    printf("%d\n",
            is_power_of_two(atoi(argv[1])));
}
```

Q. Determine if n is a power of two (continued)

- Since a power of two will only have one 1 bit, is there a bitwise operation that can determine yes/no/true/false with `n` and another operand?

```
void main(int argc, char **argv) {
    int v = atoi(argv[1]);
    printf("%d\n",
            (v & (v - 1)) == 0);
}
```

Q. Create a bitmask from bits j to k, inclusive

- j = 8, k = 4

  - Mask will be 0000 0000 1111 0000

Analysis

- What does a bitmask do?

```
void main(int argc, char **argv) {
    if (argc < 4) {
        printf("Usage: %s <value> <high bit> <low bit>\n", argv[0]);
        exit(1);
    }

    // Create mask between bits j and k, inclusive.
    int v = atoi(argv[1]);
    int j = atoi(argv[2]);
    int k = atoi(argv[3]);

    int left = ~0 << j;
    int right = (1 << k) - 1;
    int mask = ~(left | right);
    int res = mask & v;

    printf("  mask -> %d\n", mask);
    printf("result -> %d\n", res);
}
```

Q. Tic-tac-toe

Analysis

- How would you store the state?
- Consider: what is the total number of combined moves?
- Can you optimize the storage space?
- What about a bit vector?

```
+----------+
|0 | 1 | 2|
|-----------|
|3 | 4 | 5|
|-----------|
|6 | 7 | 8|
+----------+
```

1 << 0 = 1
1 << 1 = 2
1 << 2 = 4
= 7 bits

1 << 3 = 8
1 << 4 = 16
1 << 5 = 32
= 56 bits

1 << 6 = 64
1 << 7 = 128
1 << 8 = 256
= 448 bits

(Note that all the left shifts compute to powers of two.)

```
int winners[] = {
    // Across.
    7,          // 1, 2, 4
    56,         // 8, 16, 32
    448,        // 64, 128, 256

    // Down.
    73,         // 1, 8, 64
    146,        // 2, 16, 128
    292,        // 4, 32, 256

    // Diagonal.
    273,        // 1, 16, 256
    84          // 4, 16, 64
};
```

```
void play(short player, int *state) {
    char buf[2];

    printf("Your play [%c]: ", player % 2 ? 'X' : 'O');
    fgets(buf, 3, stdin);

    int move = atoi(buf);

    if (player % 2) {
        if ((*state & (1 << move)) == 0)
            *state |= 1 << move;
    } else {
        short o = *state >> 16;

        if ((o & (1 << move)) == 0) {
            o |= 1 << move;
            *state |= o << 16;
        }
    }
}
```

```
short is_winner(int *winners, int *state) {
    short k = 0;
    char winner = '\0';

    short x = *state;
    short o = *state >> 16;

    while (*(winners + k)) {
        int w = *(winners + k);

        if ((w & x) == w) {
            winner = 'X';
            break;
        }

        if ((w & o) == w) {
            winner = 'O';
            break;
        }

        k++;
    }

    ...

    return 0;
}
```

References

- btoll/howto-panic-in-a-coding-interview
- K&R
- Project Euler
- LeetCode
- Cracking the Coding Interview

FIN

Benjamin Toll
benjamintoll.com
benjam72@yahoo.com