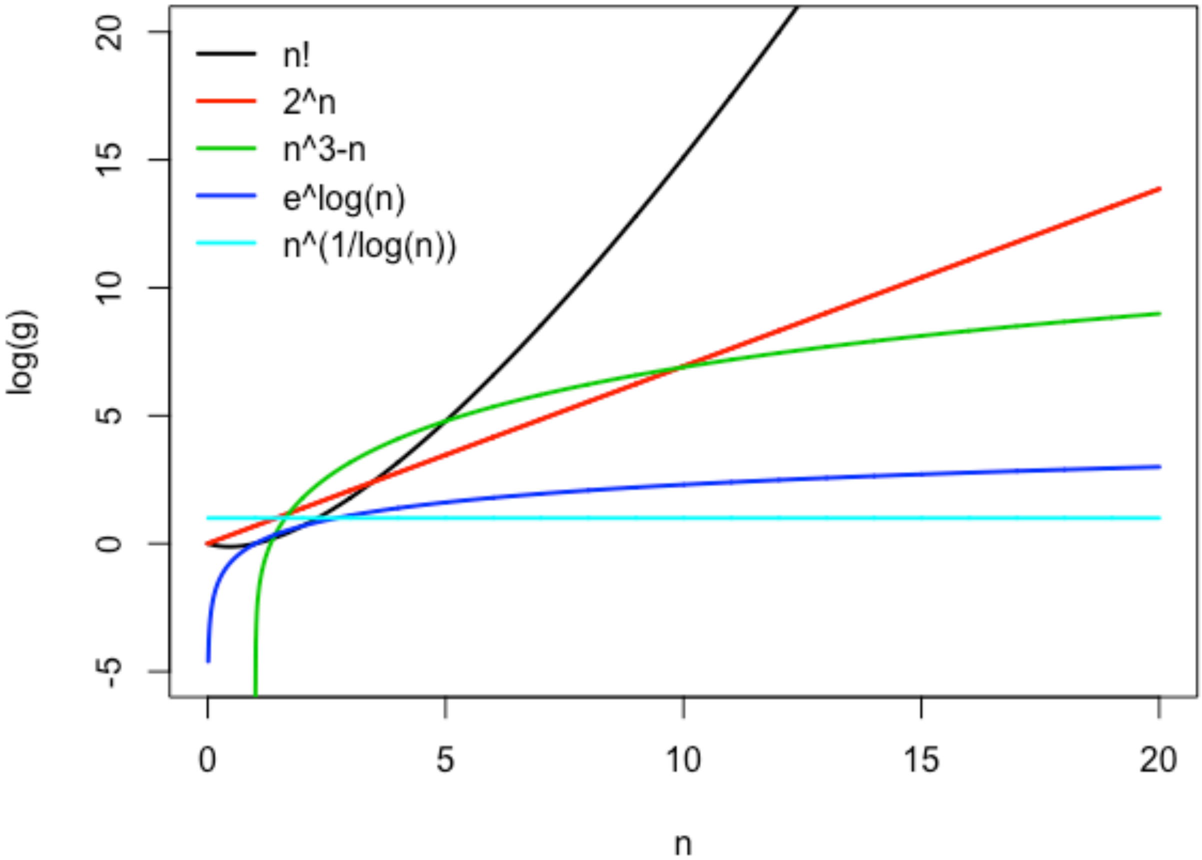
DSA Homework 1 B00401062 羅文斌

1.1

$g_1(n) = n!, g_2(n) = 2^n, g_3(n) = n^3 - n, g_4(n) = e^{\log n}, g_5(n) = n^{\frac{1}{\log n}}$

The figure shown is the log of the function $\log(g)$ against input size n.



1.2

(1) $\lim_{n\to\infty} \frac{2^n}{n!} = \lim_{n\to\infty} \frac{2}{1} \cdot \frac{2}{2} \cdot \ldots \cdot \frac{2}{k} \cdot \ldots \cdot \frac{2}{n}$. Because $\frac{2}{k} < 1$ for all $k > 2$, we can say $\lim_{n\to\infty} \frac{2^n}{n!} = 0$. Therefore, for any positive constant c, there exists a positive constant n_0 such that $0 \leq c \cdot 2^n < n!$ for all $n \geq n_0$. Therefore, $n! = \omega(2^n)$.

(2) $\lim_{n\to\infty} \frac{n!}{n^n} = \lim_{n\to\infty} \frac{n}{n} \cdot \frac{n-1}{n} \cdot \ldots \cdot \frac{n-k}{n} \cdot \ldots \cdot \frac{1}{n}$. Because $\frac{n-k}{n} = 1 - \frac{k}{n} < 1$ for all $k > 0$, we can say $\lim_{n\to\infty} \frac{n!}{n^n} = 0$. Therefore, for any positive constant c, there exists a positive constant n_0 such that $0 \leq n! < c \cdot n^n$ for all $n \geq n_0$. Therefore, $n! = o(n^n)$.

1.3

(a)

Forward: If $f(n) = O(g(n))$, meaning that there exist positive constants c, and n_0, such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$, then there also exist positive constants $\frac{1}{c}$, and the same n_0, such that $0 \le \frac{1}{c} \cdot f(n) \le g(n)$ for all $n \ge n_0$. Therefore, $g(n) = \Omega(f(n))$.
Backward: If $g(n) = \Omega(f(n))$, meaning that there exist positive constants c, and n_0, such that $0 \le c \cdot f(n) \le g(n)$ for all $n \ge n_0$, then there also exist positive constants $\frac{1}{c}$, and the same n_0, such that $0 \le f(n) \le \frac{1}{c} \cdot g(n)$ for all $n \ge n_0$. Therefore, $f(n) = O(g(n))$.

(b)

Forward: If $f(n) = \Theta(g(n))$, meaning that there exist positive constants c_1, c_2, and n_0, such that $0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for all $n \ge n_0$, then there also exist the same positive constants c_1, c_2, and n_0, such that $0 \le c_1 \cdot g(n) \le f(n)$, and $0 \le f(n) \le c_2 \cdot g(n)$ for all $n \ge n_0$. Therefore, $f(n) = O(g(n))$, and $f(n) = \Omega(g(n))$.
Backward: If $f(n) = O(g(n))$, and $f(n) = \Omega(g(n))$, meaning that there exist positive constants c_1, c_2, and n_0, such that $0 \le c_1 \cdot g(n) \le f(n)$, and $0 \le f(n) \le c_2 \cdot g(n)$ for all $n \ge n_0$, then there also exist the same positive constants c_1, c_2, and n_0, such that $0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for all $n \ge n_0$. Therefore, $f(n) = \Theta(g(n))$.

(c)

Forward: If $f(n) = O(g(n))$, meaning that there exist positive constants c, and n_0, such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$. Becuases $g(n)$ is positive, we have $0 \le f(n) \cdot g(n) \le c \cdot g(n)^2$ for all $n \ge n_0$. Therefore, $f(n) \cdot g(n) = O(g(n)^2)$.
Backward: If $f(n) \cdot g(n) = O(g(n)^2)$, meaning that there exist positive constants c, and n_0, such that $0 \le f(n) \cdot g(n) \le c \cdot g(n)^2$ for all $n \ge n_0$. Becuases $g(n)$ is positive, we have $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$. Therefore, $f(n) = O(g(n))$.

(d)

Forward: If $f(n) = O(g(n))$, meaning that there exist positive constants c, and n_0, such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$. Becuases $f(n)$ and $g(n)$ are positive, we have $0 \le f(n)^2 \le c^2 \cdot g(n)^2$ for all $n \ge n_0$. Therefore, $f(n)^2 = O(g(n)^2)$.
Backward: If $f(n)^2 = O(g(n)^2)$, meaning that there exist positive constants c, and n_0, such that $0 \le f(n)^2 \le c \cdot g(n)^2$. Becuases $f(n)$ and $g(n)$ are positive, we have $0 \le f(n) \le \sqrt{c} \cdot g(n)$ for all $n \ge n_0$. Therefore, $f(n) = O(g(n))$.

2.1

The time complexity of `Binary_Search` in the worst case is about proportional to # iterations of `for` loop × # of iterations of `while` loop, i.e. $N \cdot \log(N)$, or $O(n \cdot \log(n))$.
The time complexity of `Count_Search` in the worst case is about proportional to `2` × # iterations of `for` loop, i.e. $2 \cdot N$, or $O(n)$.
The space complexity of `Count_Search` is $N + K$, or $O(N + K)$. In the case when K is much bigger than N , the space complexity can be simplified as $O(K)$.
When K is large, say any possible integer stored as `unsigned int` in `C` language, we would have to create an array `B` of size up to 2^{32} but to find out that most of the spaces are not filled eventually. That is definitely a waste of memory space and may even cause a program to crash. In this case, `Binary_Search` would be favored instead.

2.2

(a)

```
function Brute_Force(A, N, k) {
    M = 0;
    for (i = 0; i < N; i++)
        for (j = i + 1; j < N; j++)
            if (A[i] + A[j] == k)
                M++;
    return M;
}
```

(b)

```
function Binary_Search(A, N, k) {
    sort(A);
    M = 0;
    for (i = 0; i < N; i++) {
        search = k - A[i];
        left = 0, right = N - 1;
        while (left <= right) {
            mid = (left + right) / 2;
            if (A[mid] == search)
                M++;
            else if (A[mid] < search)
                left = mid + 1;
            else if (A[mid] > search)
                right = mid - 1;
```

```
        }
    }
    return M;
}
```

2.3

```
function Count_Search(A, N, K, m) {
    B = malloc(K);
    for (i = 0; i < N; i++)
        B[A[i]] = true;
    for (i = 0; i < N; i++)
        for (j = i + 1; j < N; j++)
            if (B[m - A[i] - A[j]])
                return true;
    return false;
}
```

In the worst case when there is no such tuple satisfying $A[i] + A[j] + A[k] = m$, the outer `for` loop goes from 0 to `N-1`, and the inner `for` loop goes from `i+1` to `N-1`. Therefore, the time complexity is proportional to $(N-1)+...+2+1 = \frac{N(N-1)}{2}$, or $O(n^2)$.

3.1

[3,2,4,1,5] is valid.

Operations: PUSH 1, PUSH 2, PUSH 3, POP, POP, PUSH 4, POP, POP, PUSH 5, POP

3.2

```
function Queue_Valid(A, N) {
    Q = new Queue(N); // create an empty queue of size N
    for (i = 0; i < N; i++)
        B.enqueue(i); // enqueue sequence to queue
    for (i = 0; i < N; i++)
        if (A[i] != B.dequeue())
            return false;
    return true;
}
```

In this pseudo-code, `Queue_Valid` takes in 2 values `A` and `N`, where `A` is the sequence of interest, and `N` is the size of the sequence. It outputs `true` if it is queue-valid and `false` if not.

`Queue()`: create an empty queue of size N.

All numbers are enqueued sequentially, so the dequeued numbers should also be in increasing order, and unrelated to the order of enqueues and dequeues.

This algorithm has 2 independent `for` loop of N iterations. Therefore, the time complexiy is proportional to (# iterations of `for` loop)·2 = $2N$, or $O(n)$;

3.3

```
function Stack_Valid(A, N) {
    Q = new Queue(N); // create an empty queue of size N
    for (i = 0; i < N; i++)
        B.enqueue(A[i]); // enqueue sequence to queue
    S = new Stack(N); // create an empty stack of size N
    for (i = 1; i <= N; i++) {
        if (i <= Q.peek())
            S.push(i);
        while(S.size > 0 && Q.size > 0 && S.peek() == Q.peek())
        // if the number to be popped by S and the number to be dequeued by Q
        // are the same, then each other is eliminated
            assert(S.pop() == Q.dequeue());
        if (S.peek() > Q.peek())
            return false;
    }
    return true;
}
```

In this pseudo-code, `Stack_Valid` takes in 2 values `A` and `N`, where `A` is the sequence of interest, and `N` is the size of the sequence. It outputs `true` if it is stack-valid and `false` if not.

`Queue()`: create an empty queue of size N.

`Stack()`: create an empty stack of size N.

`Q.peek()`: peek a queue to see the number to be dequeued next.

`S.peek()`: peek a stack to see the number to be popped next.

`S.peek() == Q.peek()`: if the number to be popped by the stack and the number to be dequeued by the queue are the same, then each other is eliminated. e.g. `S = [1,2,3]` and `Q = [3,1,4,2,5]`, where `S.peek()` is 3, and `Q.peek()` is also 3, then after pop and dequeue, S and Q becomes `S = [1,2]` and `Q = [1,4,2,5]`.

If `S.peek() > Q.peek()`, the sequence is not a stack-valid, e.g. `S = [1,2]` and `Q = [1,4,2,5]`, where `S.peek()` is 2, and `Q.peek()` is 1, we can say the sequence is not a stack-valid, meaning that if both 2 and 1 are not popped yet, 2 should always be popped before 1.

In the worst case, when the sequence is stack-valid, the `for` loop check all numbers in the sequence. Because the `while` loop inside the `for` loop only loops when `S.peek() == Q.peek()`, # iterations of the `while` loop is at most N, i.e. the number of pairs of the same number, and independent of the `for` loop. Therefore, the time complexity is about proportional to # iterations of the `for` loop + # iterations of the `while` loop = $2N$, or $O(n)$.

Time complexity: $O(n)$.

3.4

```
function Queue_Valid(A, N) {
    for (i = 0; i < N; i++)
        if (A[i] != i + 1)
            return false;
    return true;
}
```

In this pseudo-code, `Queue_Valid` takes in `2` values `A` and `N`, where `A` is the sequence of interest, and `N` is the size of the sequence. It outputs `true` if it is queue-valid and `false` if not.

All numbers are enqueued sequentially, so the dequeued numbers should also be in increasing order, and unrelated to the order of enqueues and dequeues.

There is only one `for` loop of N iterations, so the time complexity in the worst case is about proportional to N, or $O(n)$.

Time complexity: $O(n)$.

3.5

```
function Stack_Valid(A, N) {
    for (i = 0; i < N; i++) {
        baseline = A[i]; // baseline stores the number to be compared with
        for (j = i + 1; j < N; j++) {
            // assure any numbers smaller than and after A[i] are arranged
            // in decreasing order
            if (A[j] < A[i] && A[j] > baseline)
                return false;
            else if (A[j] < baseline)
                baseline = A[j];
        }
    }
    return true;
}
```

In this pseudo-code, `Stack_Valid` takes in `2` values `A` and `N`, where `A` is the sequence of interest, and `N` is the size of the sequence. It outputs `true` if it is stack-valid and `false` if not.

Only one additional variable `baseline` is uesed to store the number to be compared with. Therefore, `Stack_Valid` uses only $O(1)$ additional space.

This algorithm assumes that any numbers in the stack smaller than the `baseline` will be popped in decreasing order. Take `A = [3,2,4,1,5]` as an example, 3 is the first number popped. Because both `2` and 1 are smaller than 3 and still in the stack, `2` will be popped before 1, i.e. in decreasing order.

Since there are `2` `for` loops. The outer `for` loop goes from 0 to `N-1`, and the inner `for` loop goes from `i+1` to `N-1`. Therefore, the time complexity is proportional to $(N-1)+...+2+1 = \frac{N(N-1)}{2}$, or $O(n^2)$.

Time complexity: $O(n^2)$.