1.2

b05902124

http://stackoverflow.com/questions/4999448

1.3 b05902124

2.1

b05902100

系籃學姊

2.2(b) b05902004

2.3 b05902108

3.2 b05902108

3.3 b05902108

3.4 b05902108

3.5 b05902108 / b05902120

Problem4

系籃學姊

b05902004 / b05902041 / b05902100 / b05902120 / b05902124

TA

https://openhome.cc/Gossip/AlgorithmGossip/InFixPostfix.htm

Problem5

系籃學姊

b05902041 / b05902128

1.1

$g(1) = n!$

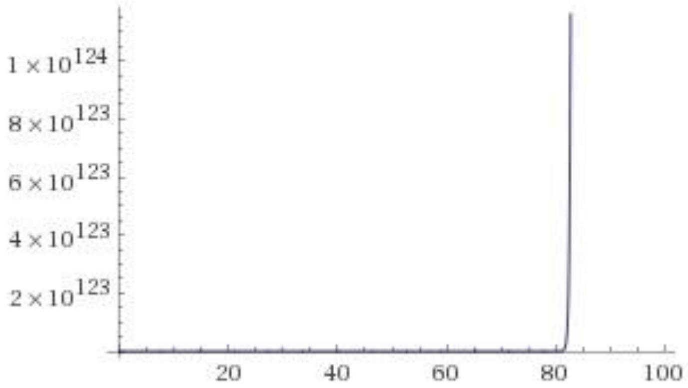
$g(2) = 2n$

$g(3) = n^3 - n$

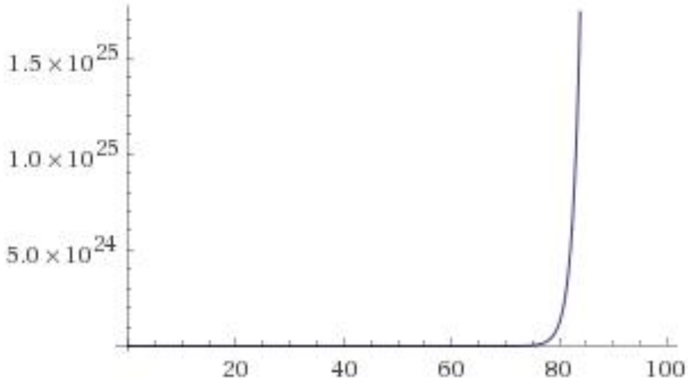
$g(4) = e^{\log n} = n$

$g(5) = n^{1/\log n} = e$

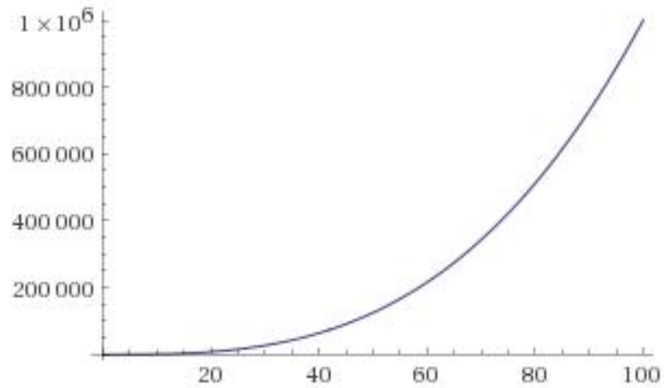
Plot: g(1)



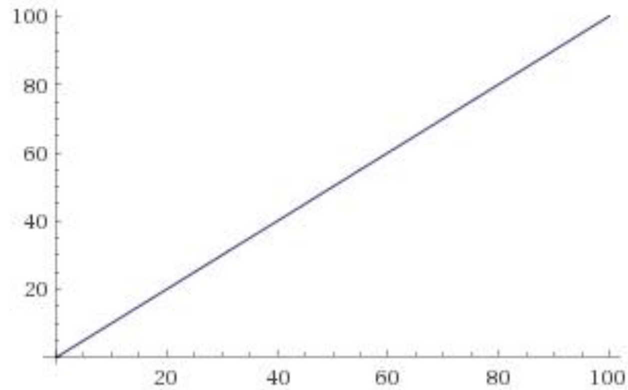
Plot: g(2)



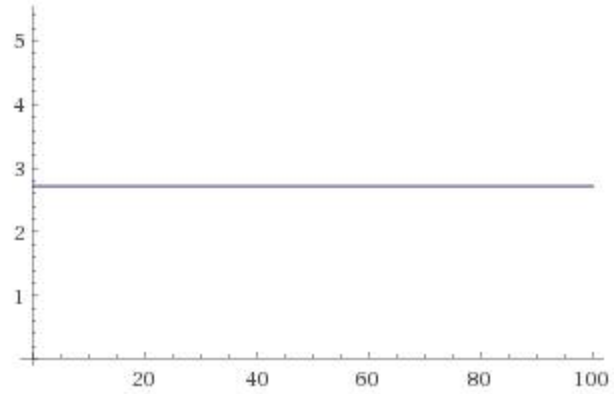
Plot: g(3)



Plot: g(4)



Plot: g(5)



1.2

$$\mathrm{f(n)} = \frac{n!}{2^n} = \frac{n(n-1)(n-2)(n-3)\ldots\ldots 3\times 2\times 1}{2\times 2\times 2\times 2\ldots\ldots\ldots\ldots 2\times 2\times 2}$$

→Both of denominator and nominator have n numbers. When n goes to infinity, f(n) goes to infinity.

→$n! > c(2^n)$

→$n! = \omega(2^n)$

$$g(n) = \frac{n!}{n^n} = \frac{n(n-1)(n-2)(n-3)......3\times 2\times 1}{n\times n\times n\times n..............n\times n\times n}$$

→Both of denominator and nominator have n numbers. When n goes to infinity, f(n) = 0.

→$n! < c(n^n)$

→$n! = o(2^n)$

1.3

(a)

f(n) = O(g(n))

→f(n) <= c*g(n)

→(1/c)*f(n) <= g(n)

→c' = 1/c，g(n) >= c' * f(n)

→g(n) =Ω(f(n))

g(n) =Ω(f(n))

→g(n) >= c*f(n)

→c' = 1/c，c'g(n) >= f(n)

→f(n) = O(g(n))

(b)

f(n) =θ(g(n))

→c_1*g(n) <= f(n) <= c_2*g(n)

→ f(n) >= c_1*g(n)、f(n) <= c_2*g(n)

→f(n) =Ω(g(n))，f(n) = O(g(n))

f(n) =Ω(g(n))，f(n) = O(g(n))

→f(n) >= c_1*g(n)，f(n) <= c_2*g(n)

→c_1*g(n) <= f(n) <= c_2*g(n)

→f(n) = θ(g(n))

(c)

$f(n) = O(g(n))$

$\rightarrow f(n) <= c*g(n)$

$\rightarrow f(n)*g(n) <= c^{*}g(n)^2$

$\rightarrow f(n)*g(n) = O(g(n)^2)$

$f(n)*g(n) = O(g(n)^2)$

$\rightarrow f(n)*g(n) <= c*g(n)^2$

$\rightarrow f(n) <= c*g(n)$

$\rightarrow f(n) = O(g(n))$

(d)

$f(n) = O(g(n))$

$\rightarrow f(n) <= c*g(n)$

$\rightarrow f(n)^2 <= c^2*g(n)^2$

$\rightarrow c' = c^2$，$f(n)^2 <= c'*g(n)^2$

$\rightarrow f(n)^2 = O(g(n)^2)$

$f(n)^2 = O(g(n)^2)$

$\rightarrow f(n)2 <= c*g(n)2$

$\rightarrow f(n) <= c1/2*g(n)$

$\rightarrow c' = c1/2$，$f(n) <= c'*g(n)$

$\rightarrow f(n) = O(g(n))$

2.1

Binary_Search(A, N, k)

sort(A) 的 time complexity 是 O(NlogN)

因為：

k 為 while 執行的次數

$N/2^k <= 1 \rightarrow N <= 2^k \rightarrow \log_2 N <= k$

$N/2^{k-1} > 1 \rightarrow N > 2^k/2 \rightarrow \log_2 N > k-1 \rightarrow \log_2 N+1 > k$

$\log_2 N <= k < \log_2 N+1$

$O(\log_2 N)$

所以：

for loop 的 time complexity 是 O(NlogN)

c_1 : sort(A)

c_2 : for loop

c_3: return

$T(N) = c_1*NlogN+c_2*NlogN+c_3$

Time complexity → O(NlogN)

Count_Search(A, N, K, k)

因為：

一個 for loop 的 time complexity 是 O(N)

c_1 : malloc

c_2 : for loop

c_3 : return

$T(N) = c_1+c_2*N+c_3$

Time complexity → O(N)

因為：

最大的 k 可能是 K+(K – 1)

所以：

最糟的情況是開一個大小為 2K 的動態陣列

Space complexity → O(K)

I think that Count_Search is the better.

By time complexity, we know O(NlogN)>O(N) when N goes to a large number.

Therefore, it will take less time than Binary_Search when N goes to a large number.

2.2

(a)

```
for i = 0 to (A.length - 1)
    for j = 0 to (A.length - 1)
        if A[i] + A[j] == k
            M++;
```

(b)

```
merge(arr, reg, 0, N)
M = 0;
for i = 0 to N - 1
    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
            right = mid - 1;
```

```
        else if A[mid] < search
            left = mid + 1;
void merge(int arr[], int reg[], int start, int end)
    if start >= end
        return;
    len = end - start + 1;
    mid = (start + end)/2;
    start1 = start;
    end1 = mid;
    start2 = mid + 1;
    end2 = end;
    merge(arr, reg, start1, end1);
    merge(arr, reg, start2, end2);
    k = start;
    while start1 <= end1 and start2 <= end2
        if arr[start1] < arr[start2]
            reg[k] = arr[start1];
            start1++;
        else
            reg[k] = arr[start2];
            start2++;
```

```
        k++;
    while start1 <= end1
        reg[k] = arr[start1];
        k++;
        start1++;
    while start2 <= end2
        reg[k] = arr[start2];
        k++;
        start2++;
    for k = start to end
        arr[k] = reg[k];
    return;
```

2.3

```
for i = 0 to N - 3
    now_k = k - A[i];
    Count_Search (A, N, K, now_k);
Count_Search (A, N, K, k) {
B = malloc (K);
for i = 0 to N - 1
    B[A[i]] = true
for i = 0 to N - 1
```

```
    if B[k - A[i]]
    return true;
return false;
}
```

3.1

push 1

push 2

push 3

pop

pop

push 4

pop

pop

push 5

pop

3.2

```
for i = 1 to n
    if A[i] != i
        return false;
    return true;
```

Time complexity → O(n)

3.3

依題意照 1, 2, 3, 4.....順序 push，若要 pop 某一個數字必定已經 push 所有比她小的數字，且遵守 stack last in first out 的規則，已 push 的數字由小到大，則 pop 出來的數字必是由大到小，所以判斷是否為 stack-valid，只要以一數為基準，被 pop 出來所有比她小的數字必須由大到小。

```
for i = 0 to n – 2
    key = A[i];
    for j = i + 1 to n - 1
        if A[j] < A[i]
            key--;
            if A[i] != key
                return false;
return true;
```

Time complexity → O(n^2)

3.4

因為：

queue-valid 的唯一可能性是[1, 2, 3, 4, 5]，pseudo code 跟 3.2 一樣

所以：

只要設一個變數 i 從 1 開始跑，一次判斷一個數字再 i++，判斷順序是不是[1, 2, 3, 4, 5]

Space complexity → O(1)

Time complexity → O(n)

3.5

因為：

設三個變數，i 以哪個數字為基準、key--數字依序遞減、j 紀錄現在的位置，開始往後跑，pseudo code 跟 3.3 一樣

所以：

再三個 space → O(1)additional space