Part 2

1 Recap

Degree of a polynomial - The highest power in the polynomial
Monotonic - Always going in one direction (either increasing or decreasing)

1.1 Logarithms

$$\log_a xy = \log_a x + \log_a y$$
$$\log_a \frac{x}{y} = \log_a x - \log_a y$$
$$\log_a x^s = s \cdot \log_a x$$
$$\log_a x = \frac{\log_b x}{\log_b a}$$

2 Asymptotics

Time complexity - Expressed in terms of the number of basic operations used by the algorithm when the input has a particular size
Worst-Case time complexity - Expressed in terms of the largest number of basic operations used by the algorithm when the input has a particular size

2.1 Big-O

Let f(x) and g(x) be functions from the set of integers or the set of real numbers to the set of real numbers. We say that f(x) is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C \cdot |g(x)|$$

whenever $x \geqslant k$

The constants C and k in the definition of big-O notation are called witnesses to the relationship $f(x)$ is $O(g(x))$

2.1.1 Example

Let $f(x) = x^2 + 2x + 1$. *Then* $f(x) = O(x^2)$

For $x \geqslant 1$, we have $1 \leqslant x \leqslant x^2$. That gives

$$f(x) = x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

for $x \geqslant 1$. Because the above inequality holds for every positive $x \geqslant 1$, using $k = 1$ and $C = 4$ as witnesses, we get

$$f(x) \leq C \cdot x^2$$

for every $x \geqslant k$

2.1.2 Example 2

Let $f(x) = 3^x$. *Then* $f(x)$ *is not* $O(2^x)$
Assume that there are constants k and C such that $3^x \leqslant C \cdot 2^x$ when $x \geqslant k$. Then

$$\left(\frac{3}{2}\right)^x \leq C$$

when $x \geqslant k$

But any exponential function a^x grows monotonically whenever $a \geqslant 1$; a contradiction

2.1.3 Sum and Product Rules

The sum rule

$$\text{If } f_1(x) \text{ is } O(g_1(x)) \text{ and } f_2(x) \text{ is } O(g_2(x)), \text{ then } f_1(x)+f_2(x) \text{ is } O\left(\max\left\{\left|g_1(x)\right|, \left|g_2(x)\right|\right\}\right)$$

The product rule

$$\text{If } f_1(x) \text{ is } O(g_1(x)) \text{ and } f_2(x) \text{ is } O(g_2(x)), \text{ then } f_1(x)\cdot f_2(x) \text{ is } O(g_1(x)\cdot g_2(x))$$

2.2 Big-Omega

Let $f(x)$ and $g(x)$ be functions from the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that

$$|f(x)| \geq C \cdot |g(x)|$$

whenever $x > k$. Note that this implies that $f(x)$ is $\Omega(g(x))$ iff $g(x)$ is $\mathcal{O}(f(x))$

2.3 Theta

Let $f(x)$ and $g(x)$ be functions from the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $\mathcal{O}(g(x))$ and $f(x)$ is $\Omega(g(x))$

This is the equivalent to saying that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $\mathcal{O}(g(x))$ and $g(x)$ is $\mathcal{O}(f(x))$

2.4 Little-o

Let $f(x)$ and $g(x)$ be functions from the set of real numbers to the set of real numbers. We say that $f(x)$ is $o(g(x))$ when

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$$

The definition without the limit is

$$o(g) = \{f : \mathbb{N} \to \mathbb{N} | \forall C > 0 \exists k > 0 : C \cdot f(n) < g(n) \forall n \geq k\}$$

This shows that $f(x)$ is $o(g(x))$ implies $f(x)$ is $\mathcal{O}(g(x))$

2.4.1 Sublinear functions

A function is called sublinear if it grows slower than a linear function. With little-o notation, we can express this as

A function $f(x)$ is called sublinear if $f(x)$ is $o(x)$, so if

$$\lim_{x\to\infty} \frac{f(x)}{x} = 0$$

2.5 Little-omega

ω is to o what Ω is to $\mathcal{O}$

$$f = \omega(g) \quad \Leftrightarrow \quad g = o(f)$$

or

$$\omega(g) = \{f : \mathbb{N} \to \mathbb{N} | \forall C > 0 \exists k > 0 : f(n) > C \cdot g(n) \forall n \geq k\}$$

2.6 General Rules

$$\text{If } f_1(x) \text{ is } o(g(x)) \text{ and } f_2(x) \text{ is o } (g(x)), \text{ then } f_1(x)+f_2(x) \text{ is } o(g(x)).$$
$$\text{If } f_1(x) \text{ is } O(g(x)) \text{ and } f_2(x) \text{ is o } (g(x)), \text{ then } f_1(x)+f_2(x) \text{ is } O(g(x)).$$
$$\text{If } f_1(x) \text{ is } \Theta(g(x)) \text{ and } f_2(x) \text{ is o } (g(x)), \text{ then } f_1(x)+f_2(x) \text{ is } \Theta(g(x)).$$

Equivalent to $\leqslant$

$$O(g) = \{f : \mathbb{N} \to \mathbb{N} | \exists C, k > 0 : \mathbf{f(n)} \leq \mathbf{C} \cdot \mathbf{g(n)} \forall n \geq k\}$$

Equivalent to $\geqslant$

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{N} | \exists C, k > 0 : \mathbf{f(n)} \geq \mathbf{C} \cdot \mathbf{g(n)} \forall n \geq k\}$$

Equivalent to $=$

$$\Theta(g) = \{f : \mathbb{N} \to \mathbb{N} | \exists C_1, C_2, k > 0 : \quad \mathbf{C_1} \cdot \mathbf{g(n)} \leq \mathbf{f(n)} \leq \mathbf{C_2} \cdot \mathbf{g(n)} \forall n \geq k\}$$

Equivalent to $<$

$$o(g) = \{f : \mathbb{N} \to \mathbb{N} | \forall C > 0 \exists k > 0 : \mathbf{C} \cdot \mathbf{f(n)} < \mathbf{g(n)} \forall n \geq k\}$$

Equivalent to $>$

$$\omega(g) = \{f : \mathbb{N} \to \mathbb{N} | \forall C > 0 \exists k > 0 : \mathbf{f(n)} > \mathbf{C} \cdot \mathbf{g(n)} \forall n \geq k\}$$

3 Sorting

3.1 Insertion Sort

Listing 1: InsertionSort ($a_1 \ldots, a_n \in \mathbb{R}, n \geqslant 2$)

```
for j=2 to n do
    x = a_j
    i=j-1
    while i>0 and a_i > x do
        a_{i+1} = a_i
        i=i-1
    end while
    a_{i+1} = x
end for
```

We know:

- When j has a certain value, it inserts the j-th element into already sorted sequence $a_1, \ldots, a_{j-1}$
- Can be proved by using invariant "after jth iteration first j+1 elements are in order"
- Running time between $n-1$ and $\dfrac{n(n-1)}{2}$ - worst case $O(n^2)$

3.2 Selection sort

Listing 2: SelectionSort ($a_1, ..., a_n \in \mathbb{R}, n \geqslant 2$)

```
for i=1 to n-1 do
    elem = a_i
    pos = i
    for j=i+1 to n do
        if a_j<elem then
            elem=a_j
            pos=j
        end if
    end for
    swap a_i and a_pos
end for
```

How does it work?
Invariant: after ith iteration positions 1,...,i contain the overall i many smallest elements in order
Not necessarily the first i elements (as it was in InsertionSort)
In the ith iteration of outer loop, we search the ith smallest element in remainder (positions $i_1, ..., n$) of input and swap it into position i

- elem keeps track of current idea of **value** ith smalllest element
- pos keeps track of the current idea of **position** of ith smallest element

Time complexity:

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 &= \sum_{i=1}^{n-1}(n-i)\\ &= \left(\sum_{i=1}^{n-1} n\right) - \left(\sum_{i=1}^{n-1} i\right)\\ &= (n-1)\cdot n - \frac{n(n-1)}{2}\\ &= \frac{n(n-1)}{2}\\ &= O\left(n^2\right)\end{aligned}$$

3.3 Bubble sort

Listing 3: BubbleSort-1 ($a_1, ..., a_n \in \mathbb{R}, n \geqslant 2$)

```
for i=1 to n-2 do
    for j=1 to n-1 do
        if a_j > a_{j+1} then
            swap a_j and a_{j+1}
        end if
    end for
end for
```

This can be improved by keeping track of whether or not an element was swapped

Listing 4: BubbleSort-1 ($a_1, ..., a_n \in \mathbb{R}, n \geqslant 2$)

```
for i=1 to n-1 do
    swaps=0
    for j=1 to n-1 do
        if a_j > a_{j+1} then
            swap a_j and a_{j+1}
            swaps=swaps+1
        end if
    end for
    if swaps ==0 then
        break
    end if
end for
```

Proof of correctness
A sequence $(a_1, ..., a_n)$ is sorted if for every adjacent pair a_i, a_{i+1} we have $a_i \leqslant a_{i+1}$
Bubble sort achieves just that

Time complexity

$$\sum_{i=1}^{n-1}\sum_{j=1}^{n-1} 1 = \sum_{i=1}^{n-1}(n-1)$$
$$= (n-1)^2 = O\left(n^2\right)$$

3.4 Mergesort

Listing 5: list MergeSort (list m)

```
if length(m) ⩽ 1 then
    return m
end if
int middle = length(m) / 2
list left, right, leftsorted, rightsorted
left = m[1..middle]
right = m[middle+1..length(m)]
leftsorted = MergeSort(left)
rightsorted = MergeSort(right)
return Merge(leftsorted, rightsorted)
```

There is then the merge function:

Listing 6: list MergeSort (list left, list right)

```
list result
while length(left)>0 or length(right)>0 do
    if length(left)>0 and length(right)>0 then
        if first(left) ⩽ first(right) then
            append first(left) to result
            left = rest(left)
        else
        #Keeping extra copies of the data in the result array
            append first(right) to result
            right = rest(right)
        end if
    else if length(left)>0 then
        append left to result
        left = empty list
    else    # Length(right) > 0
        append right to result
        right = empty list
    end if
end while
return result
```

3.5 Quicksort

Listing 7: QuickSort(int A[1..n], int left, int right)

```
if (left<right) then
    #rearrange/partition in place
    #return value "pivot" is index of pivot element in A[] after partitioning
    pivot=Partition(A,left,right)
    #Now:
    #Everything in A[left...pivot-1] is smaller than pivot
    #Everything in A[pivot+1..right] is bigger than pivot
```

```
    QuickSort (A,left,pivot-1)
    QuickSort(A,pivot_1,right)
end if
```

An example of the partition function is

Listing 8: int Partition(A[1...n], int left, int right)

```
int x =A[right]
int i=left-1
for j=left to right-1 do
    if A[j]<x then
        i=i+1
        swap A[i] and A[j]
    end if
end for
swap A[i+1] and A[right]
return i+1
```

4 Recurrences

4.1 Induction

Basically:

- "guess" correct solution (good job all sorting algorithms are n log n)
- verify base case(s) and step

Consider the recurrence for merge sort:

$$T(n) \leq \begin{cases} d & \text{if } n \leq c, \text{ for constants } c, d > 0 \\ 2 \cdot T(n/2) + a \cdot n & \text{otherwise} \end{cases}$$

To get the base case, do as follows:

$$d \leq \alpha n \log_2 n \leq \alpha c \log_2 c \quad \Leftrightarrow \quad \alpha \geq \frac{d}{c \log_2 c}$$

As for the inductive step, plug the guess in

$$\begin{aligned}
&T(n) \leq 2T(n/2) + an \\
&T(n) \leq 2\alpha \tfrac{n}{2} \log_2 \tfrac{n}{2} + an \\
&T(n) \leq 2\alpha \tfrac{n}{2} \left(\log_2(n) - \log_2(2)\right) + an \\
&T(n) \leq 2\alpha \tfrac{n}{2} \left(\log_2(n) - 1\right) + an \\
&T(n) \leq \alpha n \left(\log_2(n) - 1\right) + an \\
&T(n) \leq \alpha n \log_2(n) - \alpha n + an \\
&T(n) \leq \alpha n \log_2 n \quad \text{if } \alpha n \geq an \Leftrightarrow \alpha \geq a
\end{aligned}$$

The requirement that $\alpha \geqslant a$ suggests there isn't much room for (asymptotic) improvement, as both are constants

4.2 Iterative substitution

Expand the recurrence

$$\begin{aligned}
T(n) &\leq 2T(n/2) + an \\
&\leq 2(2T(n/4) + an/2) + an = 4T(n/4) + an + an \\
&= 4T(n/4) + 2an \leq 4(2T(n/8) + an/4) + 2an \\
&= 8T(n/8) + an + 2an = 8T(n/8) + 3an \\
&\leq 8(2T(n/16) + an/8) + 3an = 16T(n/16) + an + 3an \\
&= 16T(n/16) + 4an
\end{aligned}$$

4.3 Master Theorem

This can be used to solve recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

for $a \geqslant 1$ and $b \geq 1$

There are 3 cases, but rob says he'll give you them