Week 7 dis 1B Notes — Yiyan Chen

Big-O Analysis

Rules

① Ignore the constant multipliers
e.g. $O(10) = O(1)$, $O(cN) = O(N)$ for c a constant

② Only keep the highest order terms
e.g. $O(N+c) = O(N)$ for c a constant. $O(N(N+1)) = O(N^2+N) = O(N^2)$

Ordering ($d>0$, $p>0$, $p,d>0$, $c>1$)

$$O(1) < O(\log\log N) \leq O((\log N)^d) \leq O(N^p) < O(N^p(\log N)^d) < O(c^N) < O(N!) \approx O(N^N)$$

(log both sides; log both sides; log both sides; Stirling's Approx.)

Code exples.

①
```
for(i=0; i<N; ++i){
    O(logN)
}
```
$O(N\log N)$

②
```
for(i=1; i<N; i*=2){
    O(N)
}
```
$O(N\log N)$

③
```
for(i=1; i*i<N; ++i){
    O(1)
}
```
$O(N^{\frac{1}{2}})$

④
```
for(i=0; i<N; i+=10){
    O(1)
}
```
$O(N/10) = O(N)$

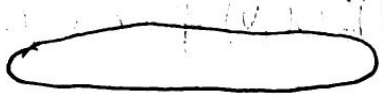
⑤
```
for(i=0; i<N; ++i){
    for(j=0; j<i; ++j){
        O(1)
    }
}
```
$O(0+1+\cdots+N-1) = O(\frac{N(N-1)}{2}) = O(N^2)$

⑥
```
for(i=0; i<N; ++i){
    for(j=0; j*j<i; ++j){
        O(1)
    }
}
```
$O(0+\sqrt{1}+\sqrt{2}+\cdots+\sqrt{N-1}) = O(\int_1^N \sqrt{x}\,dx) = O(N^{\frac{3}{2}})$

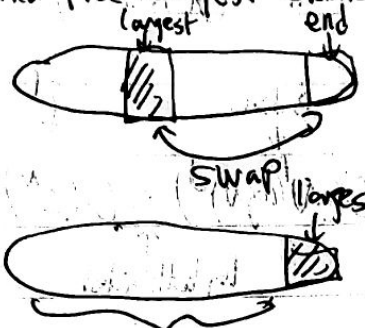
Sortings.

① Selection Sort:

Arr:



Find the largest element in the current array and put it to the end

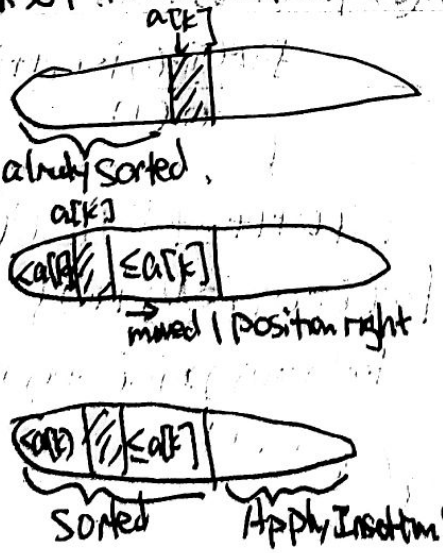


Finding largest element takes $O(k)$ time if k elements in current array.

total Complexity: $O(N+(N-1)+\cdots+1) = O(N^2)$

② Insertion Sort.

Insert the current element $a[k]$ to already sorted $a[0] \sim a[k-1]$

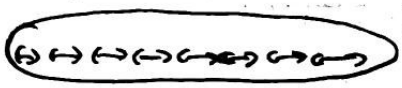


Insertion takes $O(k)$ time because of moving elements.

total Complexity: $O(1+\cdots+(N-1)) = O(N^2)$

For insertion Sort. if we know that each element is at most distance c away from their sorted locations (locality assumption), then we only need to insert them into the nearest c elements.
Sometimes, the locality assumption is worded as "the array is almost sorted"
With this property, the complexity for insertion Sort becomes
$O(cN) \approx O(N)$ if c is a constant.

③ Bubble Sort:



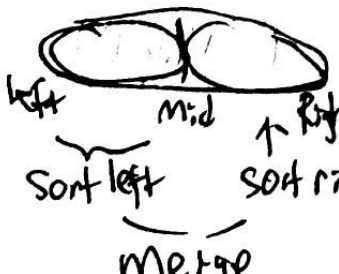
```
for(int i=0; i<k-1; ++i){
    if (a[i]>a[i+1]) swap(a[i],a[i+1]);
}
```

This guarantees that after each iteration, the largest element in the current array is swapped to the end of the array.

Apply Bubble Sort to the subarray again.

each iteration takes $O(k)$ complexity if k is the current array size.
total complexity: $O(N+(N-1)+\cdots+1) = O(N^2)$

④ Merge Sort



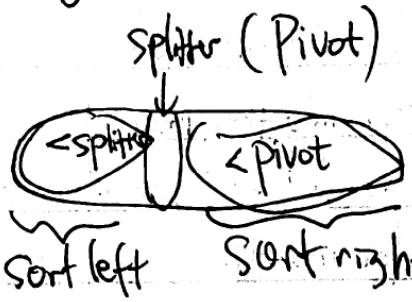
Sort left/right ↓ Merge ↓

$T(N) = 2T(N/2) + O(N)$.

total complexity: will be discussed later.

$O(N \log N)$

⑤ Quick Sort



The choice of Splitter (pivot) is somehow important. If splitter is bad.

In the worst case:

This is simply the Selection Sort.

$T(N) = T(N-1) + O(N)$ ← Complexity: $O(N^2)$

↑ Sort right ↑ split

To pick a good splitter at higher probability, one could sample some elements and pick the median. According to "law of large numbers" (Chernoff bound), with high probability that median is good.

Average case: left and right have almost even size.

Sort left&right ↓ Split ↓

$T(N) = 2T(N/2) + O(N)$ (complexity: will be discussed later.)

$O(N \log N)$

Solving the Recurrence Relation

Given recurrence relations such as,

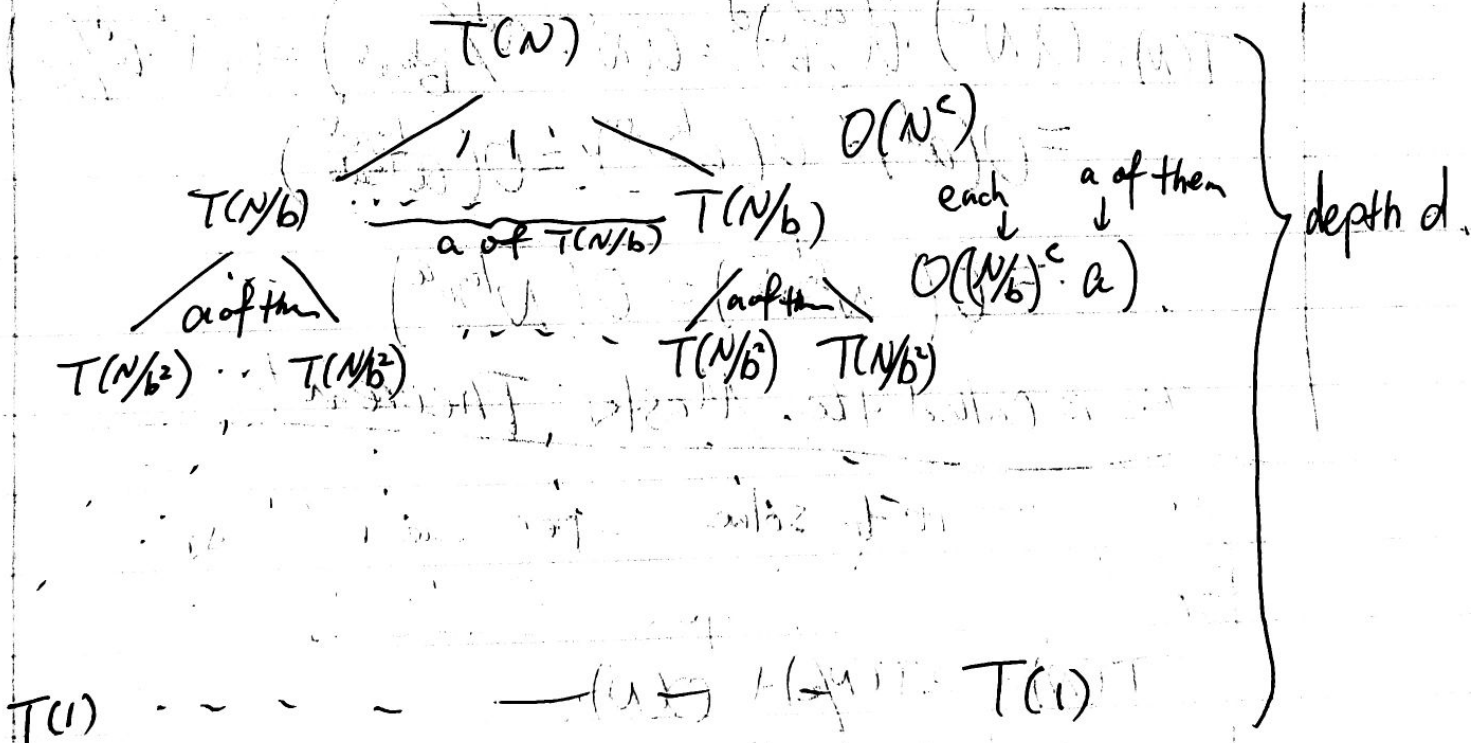
$$T(N/2) = 2T(N/2) + O(N),$$

we want to know the time complexity $T(N)$ in a closed form.

This is equivalent to finding a closed form solution to a more general form of Recurrence relations.

$$T(N) = aT(N/b) + O(N^c).$$

If we draw out the recursion tree, we have



The depth of the tree $d = O(\log_b N)$ since it reaches leaves when $b^d \approx N$.

Total complexity:

$$T(N) = O(N^c + (N/b)^c \cdot a + (N/b^2)^c \cdot a^2 + \cdots + (N/b^d)^c \cdot a^d)$$

$$= O(N^c) \cdot \underbrace{O(1 + a/b^c + (a/b^c)^2 + \cdots + (a/b^c)^d)}_{\text{geometric series}}.$$

Case 1: $a/b^c < 1$. Then 1st term dominates.

$T(N) = O(N^c) \cdot O(1) = O(N^c)$.

Case 2: $a/b^c = 1$. All terms are equal

$T(N) = O(N^c) \cdot O(d) = O(N^c) \cdot O(\log_b N) = O(N^c \log N)$

Case 3: $a/b^c > 1$. Then last term dominates.

$T(N) = O(N^c) \cdot O(a/b^c)^d = O(N^c \cdot a^d / b^{c \log_b N}) = O(N^c \cdot a^d / N^c)$

$= O(a^d) = O(a^{\log_b N}) = O(a^{\frac{\log_a N}{\log_a b}})$

$= O(N^{\frac{1}{\log_a b}}) = O(N^{\log_b a})$

This is called the Master Theorem!

Now we use it to solve the recurrence relations:

For MergeSort and Average Case QuickSort,

$T(N) = 2T(N/2) + O(N)$.

here $a=2, b=2, c=1$.

$a/b^c = 1$. $\therefore$ It's Case 2.

$\therefore T(N) = O(N^1 \cdot \log N) = O(N \log N)$