Basic Algorithms - CLASS 2 - 2019/9/9

Last time:

- History & background
- Insertion sort

Today:

- Big-O notation
- Merge sort, and divide & conquer.

Recap:

- Insertion sort:
  - Correctness using loop invariants
  - Run time analysis:

$$\sum_{j=2}^{n}(c+t_j) = c\cdot(n-1) + \sum_{j=2}^{n} t_j \leq c\cdot(n-1) + \sum_{j=2}^{n} j =$$

$$= c\cdot(n-1) + \frac{n(n+1)}{2} - 1$$

$$= \frac{1}{2}n^2 + (c+\frac{1}{2})\cdot n - (c+1)$$

$$= \Theta(n^2)$$

In worst case, $\Theta(n^2)$. Also in average case (since then t_j is on average $j/2$).

- It is an <u>in-place</u> algorithm (only constant extra memory needed).
- It is also stable.

(2)

Big-O notation

Often we don't want to worry about constant factors.

Cheat sheet: Θ =

O $\leq$

Ω $\geq$

o $<$

ω $>$

Def: For a function $g: \mathbb{N} \to \mathbb{R}^+$, define the sets

$$\Theta(g(n)) = \{ f: \mathbb{N} \to \mathbb{R}^+ : \exists\, c_1, c_2 > 0 \;.\; \exists\, n_0 \geq 1 \text{ s.t. } \forall n > n_0,\; c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}$$

$$O(g(n)) = \{ f: \mathbb{N} \to \mathbb{R}^+ : \exists\, c > 0\; \exists\, n_0 \geq 1 \text{ s.t. } \forall n > n_0,\; f(n) \leq c \cdot g(n) \}$$

$$\Omega(g(n)) = \{ f: \mathbb{N} \to \mathbb{R}^+ : \exists\, c > 0,\; \exists\, n_0 \geq 1 \text{ s.t. } \forall n > n_0,\; f(n) \geq c \cdot g(n) \}$$

Ex • Polynomials: $2n^2 + 5n + 10 \in \Theta(n^2)$

(We will often write $= \Theta(n^2)$)

• $n \log n = O(n^2)$ (in fact, $n \log = o(n^2)$)

• $2^n \cdot \log n \notin O(2^n)$ • $3^n \notin O(2^n)$ • $3^n + 2^n \in \Theta(3^n)$

Claim $f \in O(g)$ iff $g \in \Omega(f)$

Proof $\Rightarrow$ Assume $f \in O(g)$. Let c, n_0 be s.t. $\forall n > n_0$, $f(n) \leq c \cdot g(n)$.

Then $\forall n > n_0$, $g(n) \geq \frac{1}{c} f(n)$. Therefore $g \in \Omega(f)$.

$\Leftarrow$ Similar. ∎

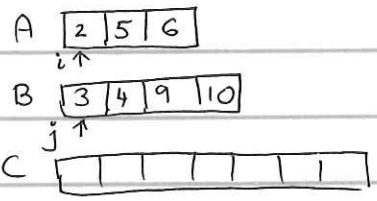
(3) Merge sort

Use divide & conquer. (Gauss 1805?)

MERGESORT (A[1..n])

- MERGESORT(A[1..$n/2$])
- MERGESORT(A[$n/2$+1..n])
- MERGE A[1..$n/2$] and A[$n/2$+1 .... n]

```
MERGE(A, m, B, n, C)   // A[1..m] and B[1..n] are sorted arrays
    i = 1
    j = 1
    FOR k = 1 to m+n
        IF A[i] ≤ B[j]
            C[k] = A[i]
            i = i+1
        ELSE
            C[k] = B[j]
            j = j+1
```



(4)

Correctness:

Loop invariant:

At the start of the for loop, $C[1 \ldots k-1]$ contains the $k-1$ smallest elements of A & B in sorted order, and these elements are $A[1..i-1]$ and $B[1..j-1]$.

Initialization: $i=j=k=1$, so the invariant holds trivially.

Maintenance: assume invariant holds for loop k, and that $A[i] \leq B[j]$ (other case similar). Then $A[i]$ is smallest among elements not yet copied to C. Therefore invariant holds in next iteration.

Termination: when $k = m+n+1$, we have $i = m+1$, $j = n+1$, and so C contains all the elements of A and B in sorted order.

Run time analysis: $\Theta(m+n)$

Back to MERGESORT..

Correctness follows from that of MERGE.

Runtime: $T(n) = 2\,T(\frac{n}{2}) + \Theta(n)$ (recursive call; merge) (Recurrence)

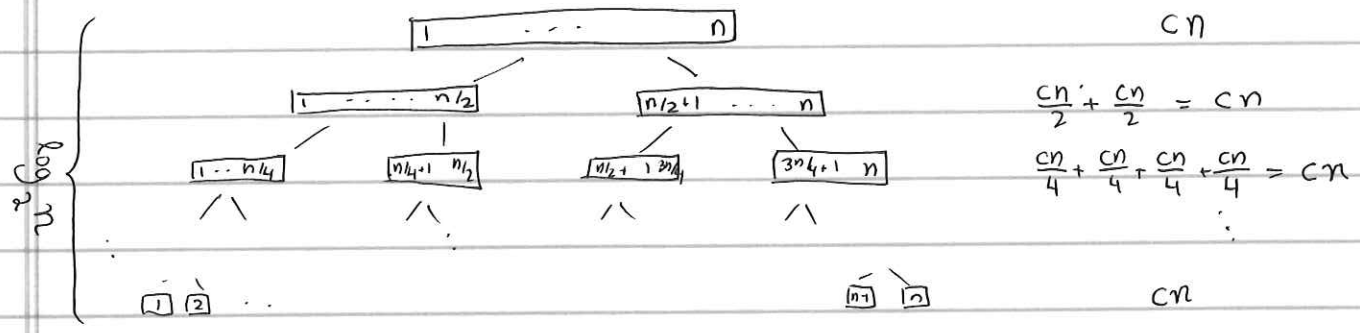
We'll show by induction that runtime is $\Theta(n)$.

Indeed, $2 \cdot \Theta(\frac{n}{2}) + \Theta(n) = \Theta(n)$.

In fact, as we will show later, sorting requires $\Omega(n \log n)$ (in comparison model).

(2)

Recursion tree: (informal, but very helpful) $T(n) = 2T(n/2) + n$



So overall runtime is $\Theta(n \log n)$.

To prove it formally, use the "substitution method" (basically induction):

Claim For large enough constant c, $\forall n \geq 2$, $T(n) \leq cn \log n$.

Proof Assume by induction true for numbers $< n$. Then,

$$T(n) \leq 2\left(c \cdot \frac{n}{2} \cdot \log \frac{n}{2}\right) + n$$

$$= c \cdot n \log \frac{n}{2} + n$$

$$= cn \log n - cn + n \leq cn \log n$$

for $c \geq 1$, as required. ∎

WRONG:

"Claim" $T(n) = O(n)$

"Proof" By induction:

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2O(n) + n = O(n).$$

BASIC ALGORITHMS - CLASS 3 - 2019/9/11

Last time:

- Big-O notation
  - We use "=" even though it's not an equality. $n = O(n^2)$, $n^2 = \Omega(n)$
  - Θ ($=$) O ($\leq$) Ω ($\geq$) o ($<$) ω ($>$)
- Insertion sort: $\Theta(n^2)$ worst case and average case
  - in place
  - stable
- Merge sort: $\Theta(n \log n)$ worst case and average case
  - Asymptotically optimal in comparison model
  - Not "in place"
  - Stable
  - Based on divide-and-conquer
  - In practice, recursive calls use indices into array, do not copy subarray (so we are "in place")
  - ① Divide ② Conquer ③ Combine

Today:

- Solving recurrence equations
- Quick sort [CLRS §7]

③ Quick-sort:

- Run time is $\Theta(n^2)$ in worst case, but $O(n \log n)$ on average (so asymp. slower than merge sort in worst case)
- In place (unlike merge sort) • Not stable
- The constant in the $O(\cdot)$ is quite low and so runs very well in practice.
- Also based on divide and conquer.
- A bit like "merge sort in reverse"

Idea:

DIVIDE
1. Select a pivot (several ways of doing it, each with own advantage)
2. Partition: everything smaller than pivot goes left, larger to the right

CONQUER
3. Recursively solve both halves

QUICKSORT(A[1...n])

1. k = PARTITION(A, [redacted]) // [redacted] k is the (final) location of the pivot after partition
2. QUICKSORT(A[1..k-1])
3. QUICKSORT(A[k+1..n])

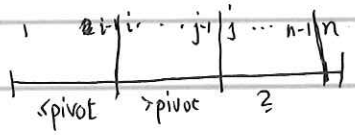
(4)

```
PARTITION ( A[1..n])      // moves a certain element "pivot" to its final location,
                          // and all other elements either left or right of it
                          // depending on whether they are smaller or bigger.
                          // Returns the final location of the pivot.
1. pivot = A[n]
2. i = 1
3. FOR j = 1 to n-1
4.     IF A[j] ≤ pivot
5.         SWAP A[i] with A[j]
6.         i = i+1
7. SWAP A[i] with A[n]
8. RETURN i
```

Correctness of PARTITION:

Loop invariant:

1. For all $k \in \{1, .., i-1\}$, $A[k] \leq$ pivot
2. For all $k \in \{i, .., j-1\}$, $A[k] >$ pivot



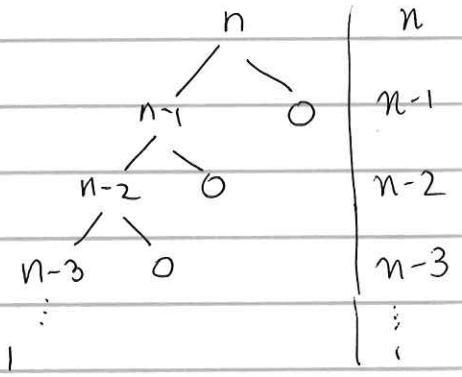
At the end of the for loop, and Step 7,
$A[1] \ldots A[i-1] \leq$ pivot
$A[i] \ldots A[n-1] >$ pivot.
and $A[i] =$ pivot.

Runtime of PARTITION: $O(n)$

⑤ Correctness of QUICKSORT follows.

Runtime?

Worst case is when array is already sorted (!).
Then each call to PARTITION split the array into $n-1$ and 0.
Since each call to PARTITION takes linear time,
overall runtime is $c \cdot \sum_{i=1}^{n} i = \Theta(n^2)$.



Best case: we split the array exactly in half. Then:

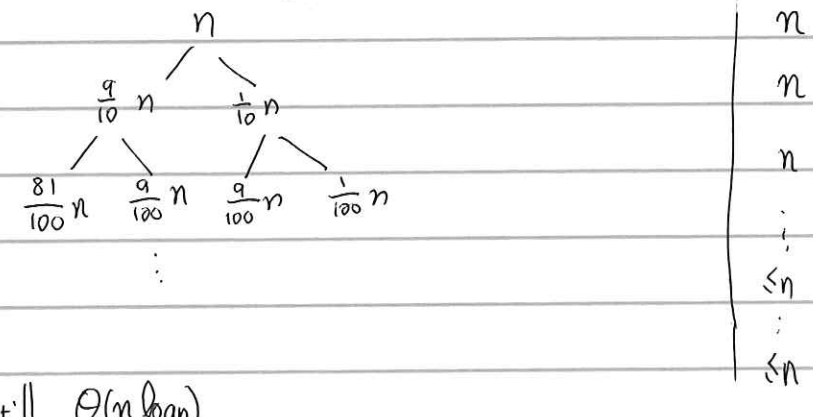
$$T(n) = 2T(\tfrac{n}{2}) + \Theta(n)$$

and so $T(n) = \Theta(n \log n)$ as we saw before.

Average case (roughly): if the array we receive is randomly shuffled, we can expect the last element to be not at the top or bottom percentiles. E.g., there is 80% chance that it is between 10% – 90% percentile. Assume for simplicity we always get a 10%/90% split:

$$T(n) = T(\tfrac{9}{10}n) + T(\tfrac{1}{10}n) + \Theta(n)$$

$\log_{\frac{10}{9}} n \leq 7 \log_2 n$
$= O(\log n)$



So: still $\Theta(n \log n)$

(6)

The algorithm we showed performs very poorly on some inputs.
One way to overcome this is to choose pivot randomly (so we get a randomized algor!).
This way, with probability 80%, we get a partition that is
at least $(\frac{1}{10}, \frac{9}{10})$ balanced, and so the expected running
time is $\Theta(n \log n)$ for all inputs (i.e., in worst case).

In practice, one can also pick the middle element, or
the median of {first, middle, last}., or the overall median.

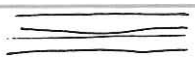
$N + \frac{N}{2} + \frac{N}{4} + \cdots + 1 \leq 2N = \Theta(N)$

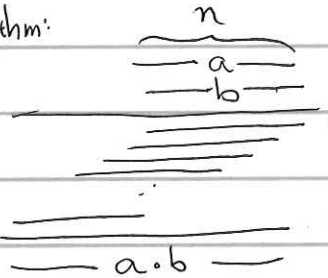
(7) Integer multiplication (Karatsuba)

Another nice example of divide & conquer.

Input: two n-bit numbers: $a = a_1 a_2 \ldots a_n$, $b = b_1 \ldots b_n$ $\quad a, b < O(2^n)$

Goal: compute (the binary representation of) $a \cdot b$.

Addition  $\Theta(n)$ runtime.

Elementary school algorithm:  $\Theta(n^2)$ runtime.

$a \cdot b = \underbrace{a + a + \cdots + a}_{b \text{ times}}$: $O(2^n \cdot n)$ runtime

Let's use divide & conquer. Write:

$$a = (a_1 \ldots a_{n/2}) 2^{n/2} + (a_{n/2+1} \ldots a_n) = a_h \cdot 2^{n/2} + a_\ell \quad (\text{as in: } 152,799 = 152 \cdot 1000 + 799)$$

$$b = (b_1 \ldots b_{n/2}) 2^{n/2} + (b_{n/2+1} \ldots b_n) = b_h \cdot 2^{n/2} + b_\ell$$

Now,

$$a \cdot b = (a_h \cdot 2^{n/2} + a_\ell) \cdot (b_h \cdot 2^{n/2} + b_\ell) =$$

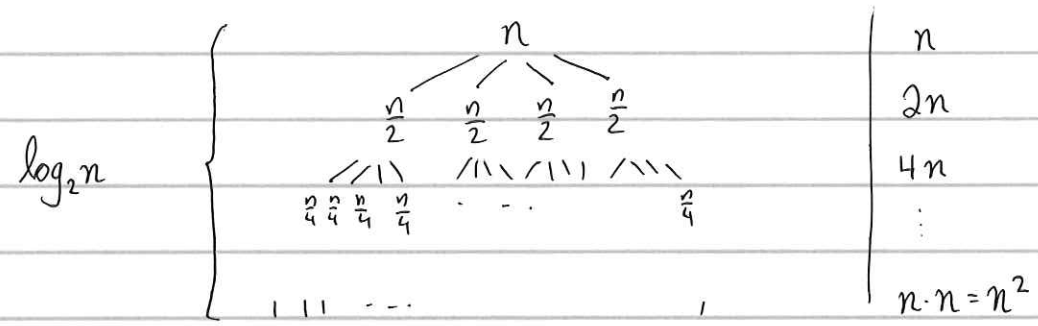
$$(a_h \cdot b_h) \cdot 2^n + (a_h \cdot b_\ell + a_\ell \cdot b_h) \cdot 2^{n/2} + (a_\ell \cdot b_\ell). \quad (*)$$

So we get:

MULT (a, b)

1. Compute MULT(a_h, b_h), MULT(a_h, b_ℓ), MULT(a_ℓ, b_h), MULT(a_ℓ, b_ℓ)
2. Output $(*)$

Time: $T(n) = \overset{(1)}{4T(\tfrac{n}{2})} + \overset{(2)}{\Theta(n)}$



Total runtime: $\Theta(n^2)$.

Idea: compute:

(1) $(a_h + a_\ell) \cdot (b_h + b_\ell) = a_h b_h + a_h b_\ell + a_\ell b_h + a_\ell b_\ell$

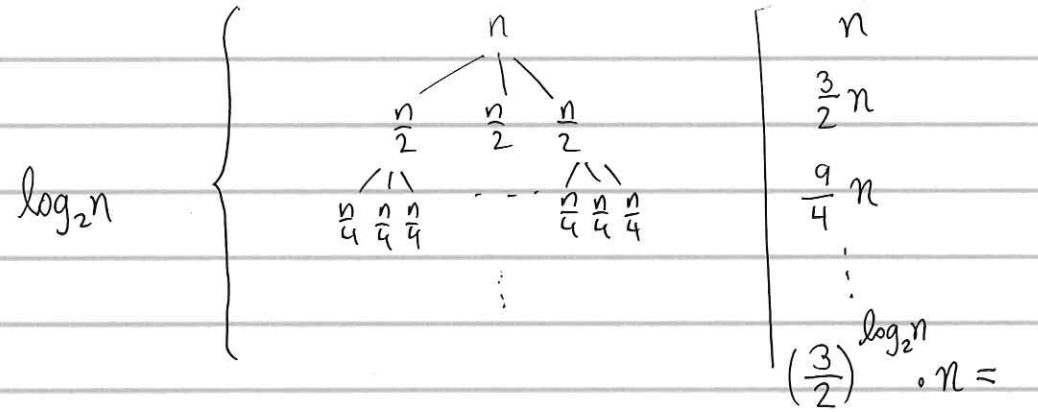
(2) $a_h \cdot b_h$

(3) $a_\ell \cdot b_\ell$

Then $a_h \cdot b_\ell + a_\ell \cdot b_h = (1) - (2) - (3)$.

We can therefore compute $a \cdot b$ using 3 multiplications (recursive calls) and 6 additions/subtractions. Therefore, the runtime is

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + 6 \cdot \Theta(n) = 3 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$$



$2^{\log_2 3/2 \cdot \log n} \cdot n =$

$n^{\log_2 3/2} \cdot n = n^{\log_2 3} = O(n^{1.585})$

So overall runtime is $O(n^{1.585})$.

Remark Toom-Cook: $T(n) = 5T(n/3) + \Theta(n)$, $\Theta(n^{\log_3 5}) = O(n^{1.465})$.

Remark Best known is $\Theta(n \log n)$ (from 2019!)

(Schönhage-Strassen '71 $\Theta(n \log n \log\log n)$ GMPY, Fürer 2007 $\Theta(n \log n \, 2^{O(\log^* n)})$ $\rightarrow 2^{64}$-digits 10,000+ digits

Closest pair of points

Problem: given n points $p_1, p_2, \ldots, p_n \in \mathbb{Z}^2$ in the plane, find a pair with smallest Euclidean distance between them.

Fundamental geometric primitive, with lots of applications (air traffic control, finding near duplicates, clustering, nearest neighbor)

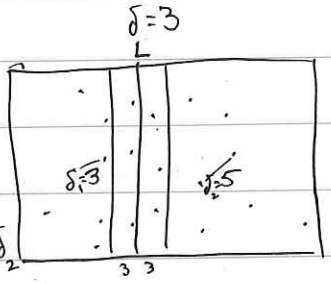
- Brute force: check all pairs of points p_i, p_j.
  Need $\binom{n}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$ distance calculation: $dist(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$.

- If points are on a line (i.e., in $\mathbb{Z}$) easy in time $O(n \log n)$.

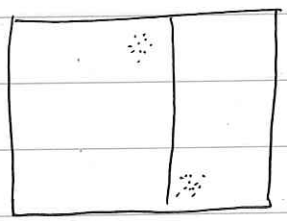
- For simplicity assume no two points have same x coordinate.

ALGORITHM:

1. Divide points into two halves using a vertical line L (i.e., based on their x coordinate)
2. Conquer: find closest pair in each side recursively, δ_1, δ_2
3. Combine: find closest among pairs crossing the line.
   Output smallest of the three numbers.



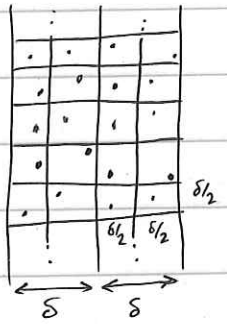
But solving Step 3 seems to require $\Theta(n^2)$ by trying all pairs!



Observation: only need to consider pairs closer than the closest distance found inside the two halves, $\delta = \min(\delta_1, \delta_2)$

Compute "cross-line" closest pair:

1. Take points within δ of L,
2. Sort these points by their y coordinate, $s_1, s_2, \ldots, s_k$
3. For each point compute its distance to the points that are not more than 11 away in the list.
4. Output closest pair found.



Claim: If $|i-j| > 11$, the distance between s_i and s_j is $\geq \delta$.

(actually 6 are enough) instead of 11

Proof:
- At most one point in each $\frac{1}{2}\delta \times \frac{1}{2}\delta$ box.
- Two points at least two rows apart have distance $\geq 2 \cdot (\frac{1}{2}\delta) = \delta$.

Correctness ✓

Runtime:

$$T(n) = 2 \cdot T(n/2) + \Theta(n \log n) \Rightarrow T(n) = \Theta(n \log^2 n).$$

Remark: Can improve to $T(n) = \Theta(n \log n)$ by only sorting once in the beginning, so that $T(n) = 2T(n/2) + \Theta(n)$