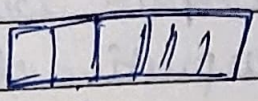
08/01/24 Lecture-2

Sorting



Input: Array A → n int.g.
output: integers arranged in ascending order

Ideas:-

(i) compare pairwise & swap ✓ (insertion sort) ✓

▷ # steps $\approx \sum_{i=1}^{n} i \approx n^2/2$ (slow algo)

(ii) // Divide & conquer (merge): Divide into halves, sort each, merge the two (merge sort)

▷ # steps $\propto t(n) = 2 \cdot t(n/2) + O(n) \approx n \cdot \log n$

Ex: $n = 10^7$: $n^2 = 10^{14} \times 1\text{ns} = 10^5\text{s} > 28\text{hrs}$.

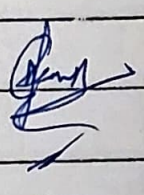
$n \lg n$

Data structure:-

(ds)

1-d array
sorted array is even more better ds.

Ex① directory/dictionary



→ $n = 10^8$ words

WAP: Input: dictionary A & string s
output: yes iff $s \in A$

→ (i) Sequential search in A
▷ # steps $\approx n \approx 10^8 \times 10^{-9}$ s $\approx$ 0.1 s / query

(ii) Binary Search

Problem – Range-Minima

WAP:

input: array A of n integers
output – A fast ds to ans. queries of the type
: Range-min. (i,j)
find the min of A[i], A[i+1], ... A[j]

Eg. A = [12, 13, 46, 54, 115]
i = 2, j = 4

Ideas:- (i) sort A in n log n time.
step = 1

(ii) Range-min brute (i,j)
steps → n

(iii) Range-min matrix A (i,j)

▷ time & HDD-space ×

ds for solving this Rng-min in eff. time & space –?

10/02/24 Lecture -3 D. ES0207

→ we saw It Fib(n) takes n steps and the same space.

Q. Find a superfast algo. for F(n) mod 2024 ?
m := 0

(i) It Fib (n,m) : for (i = 2 to n)
F[i] = F[i-1] + F[i-2]
mod m;

▷ It Fib (n,m) requires > 3n instructions
↑
Iterative fib
& only
n · log n - Space

(ii) Rec Fib (n,m) : If (n ≥ 2) return Rec Fib(n-1, m)
↑ + Rec Fib(n-2, m)
Recursive fib. mod m
else return n;

(1.5/n)

▷ Rec Fib (n,m) requires > F(n) steps > 1.5^n
[exp. time]

(iii) clever insight : DO double instead of +1!

~~take two steps instead of 1~~

$$\begin{pmatrix} F(i) \\ F(i-1) \end{pmatrix} = \begin{pmatrix} F(i-1) \\ F(i-2) \end{pmatrix}$$

$$\begin{pmatrix} F(i) \\ F(i-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F(i-1) \\ F(i-2) \end{pmatrix}$$ [Evolution of F as a 2×2 matrix]

$$= A^2 \begin{pmatrix} F(i-2) \\ F(i-3) \end{pmatrix}$$

$$A := \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$= \cdots A^{i-1} \begin{pmatrix} F(1) \\ F(0) \end{pmatrix}$$

$$= A^{i-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

▷ $F(n) = (A^{n-1})_{1,1}$

▷ $F(n) \equiv (A^{n-1} \bmod m)_{1,1}$

Insight of Squaring: Don't multiply when you can Square!

eg- A, A^2, A^4 → $A^{2^k} \bmod m$.

[Repeated Squaring]

▷ Get to 2^k- power in k- steps.

▷ Get to n^{th} in $\lceil \log n \rceil$ steps

▷ $B \times C \bmod m$ takes $(4^2 + 4)$ instruct"s.

⇒ $20 \log n$ steps.

⇒ $A^{2^k} \bmod m$ takes $20k \log n$ steps.

- cleverFib(n,m): $S[0] \leftarrow A := \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$; // A^{2^0}

$k \leftarrow \lceil \log(n-1) \rceil$;

. write $n-1 \rightarrow b_0 + b_1 2 + b_2 2^2 + \cdots + b_{k-1} 2^{k-1}$
// $n-1$ in binary

- for $(i = 1$ to $k-1)$
$S[i] \leftarrow S[i-1]^2 \bmod m$ // rep. squaring

→ $B \leftarrow S[0]^{b_0} \times S[1]^{b_1} \times \cdots \times S[k-1]^{b_{k-1}} \bmod m$
// $B = A^{b_0 + 2b_1 + \cdots + 2^{k-1} b_{k-1}}$
$= A^{n-1} \bmod n$

→ return $B_{1,1}$;
// $B_{1,1} = F(n) \bmod n$

▷ cleverFib(n,m) takes only $\leq 2 \log m * \log n \times 2$
$= 4 \log m * \log n$ steps
$= 4 \log m \cdot \log n$ steps & $\log m \cdot \log n$ space (array)

(24)

logarithmic << linear << exponential
(clever) (iterative) (recursive)
$(\log n)$ (n) (2^n)

n → input size

Time complexity of an algorithm

↳ no. of instructions required in the worst case, as a function of input size. (n)

- Detour into multiplication $a * b \bmod m$:
  takes $\approx (\log m)^2$ time $\approx \log m$ bits

→ Advanced algos takes $\approx \log m$ time

	Algos	Input-size	Complexity
(1)	1^{st} Fib(n, m)	$\log m + \log n$	$\approx n \log m$ (exp)
(2)	cleverFib(n, m)	$\log m + \log n$	$\approx 40 \log m \log n$ [quad]
(3)	Sort $A[0 \ldots n]$	n words	$\approx \frac{n^2}{2}$; $\approx n \log n$ [almost linear]
(4)	Test if $A[0 \ldots n]$	n words	$\approx n$

Issue in comparison : $40 \log n > 3n$ for small n!

→ But, $40 \log n << 3n$, for all $n \geq n_0 := 128$.

→ ignore small n's, but ~~look~~ focus on growing n.

→ you should compare the asymptotics of functions in n; $n \to \infty$

Eg: Algo. A runs in time $t_A(n) = 5n^2 + n + 10$
Algo. B —"— $t_B = n^2 - 1$
Algo. C $t_C = 100 n^{1.5} + 1000$

→ C is a better improvement over A & B
B is similar to A.

▷ $\lim_{n\to\infty} \frac{t_B}{t_A} \to \frac{1}{5}$

▷ $\lim_{n\to\infty} \frac{t_C}{t_A} \left(= 100\, n^{-0.5}\right) \to 0$

[Caveat: Beware of astronomical constants in practice]

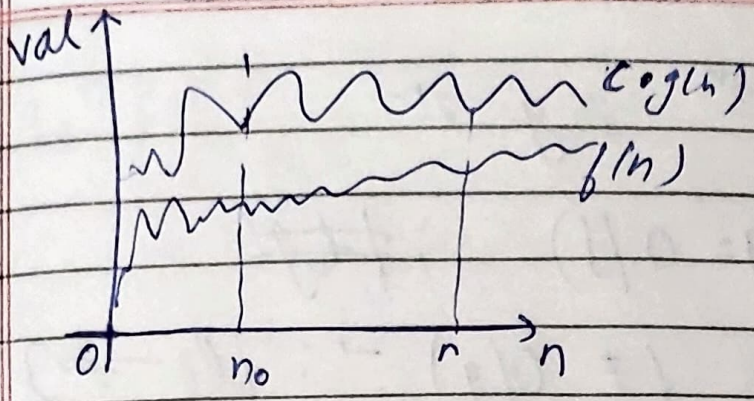
While comparing:

Obs. 1: Ignore additive/multiplicative constants
Obs. 2: Identify the leading monomial in n.

Order notation

⇒ Functions $f, g : \mathbb{N} \to \mathbb{R}$
(time) $\nearrow n \to f(n), g(n)$

$f(n)$ is of the order of $g(n)$ if
$\exists\, c, n_0 \in \mathbb{R} > 0$,
$\forall\, n \geq n_0$, $f(n) \leq c \cdot g(n)$



write ↳ $f = O[g]$ (Bigoh]
time complexity

eg→ $5n^2 + n + 10 = O[n^2]$

$100n^{1.5} + 1000 = O(n^{1.5})$ → tight/optimal

↳ $= O(n^2)$ ← loose

$100 = O(1)$ not $[O(1) \neq 100]$

- In this course, we'll $g(n)$ that is tight

Proposition:

(i) $f = O(g)$, $g = O(h) \Rightarrow f = O(h)$

(ii) $f, g = O(h) \Rightarrow f+g = O(h)$
$f \cdot g = O(h^2)$

(iii) $f = O(h)$, $g = O(1) \Rightarrow f \cdot g = O(h)$

Eg. $f = 3^n$; $g = 2^n$

$g = O(f)$; ~~$f = g$~~

$f \neq O(g)$ $\rightarrow$ $\because [\forall g \to \infty]$

(one sided upper bound)

Eg. $t_c = O(t_B)$; $t_B \neq O(t_c)$

t_c is better (smaller order) than t_B

↓

c is better

$\Rightarrow$ Asymptotics:

▷ Insertion-sort : $O(n^2)$ time

merge-sort : $O(n \log n)$ time

binary search : $O(\log n)$ time

clever fib : $O(\log n \log m)$ time

→ Designing (asymptotically) fast algos.

↳ ~~art~~ art.

15/01/24

- Let's consider some challenging problems in algorithm design
- problem: Max-Sum Subarray

WAP: Input: Array A storing n integers
output: Subarray $B = A[i \cdots j]$ with max possible sum.

Ideas: (i) Max-Sum-Brute (A): for $i, j = 0 \cdots n-1$ {
compute sum $A[i \cdots j]$;
Track the maximum-sum i, j in max;}

▷ # steps $\approx \sum_{i,j=0}^{n-1} (j - i + 1) \leq O(n^3)$

Qn: reduce it to linear-time $O(n)$?
- what if we reduce the number of indices from 2 to 1? $(i, j) \mapsto j$?
↳ Reduce the search-space

▷ Defn: $s[j]$:= Sum of the max-sum subarray ending at j.

Eg: A: 3, -5, 3, 8, 2, -4 (↓ $j=5$)
1 1 1 1 1 1
3 -2 3 11 13 9

$s[j] \rightarrow$

▷ If we have $s[0 \cdots n-1]$ then we can output max-sum subarray of A in $O(n)$ time.

Pf: output $\max(s[0 \cdots n-1])$ = max-sum subarray of A. (∵ each j appears in s.)

Q compute $s[j]$ in $O(1)$ time; given the earlier value;
⇒ gives an $O(n)$ time algo!

▷ If $B = A[i \cdots j]$ achieves the optimum $s[j]$, then there are two options:

$B' \cup \{A[j]\}$, where $B' := A[i \cdots j-1]$ achieves the sum $s[j-1]$

$A[j]$

X (no 3rd method)

Pf. To get $S[j]$ either you use a prefix before $A[j]$, or you don't.

→ the latter if done $\Leftrightarrow S[j-1] < 0$.

Lemma: (update): $S[j-1] > 0 \Rightarrow S[j] = S[j-1] + A[j]$,
$\nexists\ S[j-1] \leq 0 \Rightarrow S[j] = A[j]$.

Pf. In case-1: $A[i \cdots j-1]$ extends to $A[i \cdots j]$.
In case-2: $A[i \cdots j-1]$ is modified completely to $A[j]$

Exercise: use this to calculate the $I[\,]$ array values such that $I[j] = i$ with $S[j]$ = sum of $A[i \cdots j]$

Max-Sum-Subarray ($A[0 \ldots n-1]$): $S[0] \leftarrow A[0]$;

$O(n)$ { • for $j = 1 \ldots n-1$ {
if ($S[j-1] > 0$) $S[j] \leftarrow S[j] + A[j]$;
else $S[j] \leftarrow A[j]$;
}

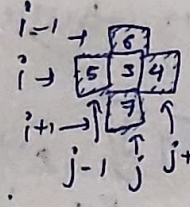
$O(n)$ { • Find max $S[j]$ // say, $S[j_0]$
• output $(j_0, S[j_0])$;

▷ time $\approx O(n)$

Problem: local-Minima in a Grid

WAP: Input: $G[n][n]$ storing $n \times n$ distinct integers

output: Find a local-minima, i.e. an entry $G[i][j]$ smaller than its four neighbours.



▷ A local-minima exists in G.

Pf. ∵ Global-Minima is also local-minima.

Idea (i) Single-scan of G yields a local-minima.
Time $\approx O(n^2)$.

Q. can you reduce it? $O(n)$?

(ii) Local exploration / iteration: Go to a smaller neighbor, and repeat!
↳ If you can't, then output the current cell.

```
Local Explore (G[1..]):   c := G[0][0];
  • while (c is not local-minima)   c ← least neighbor;
  • return c;
```

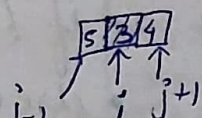
▷ No cell is visited twice (in algo.)

Pf. value c decreases in every step.
⇒ while-loop halts.

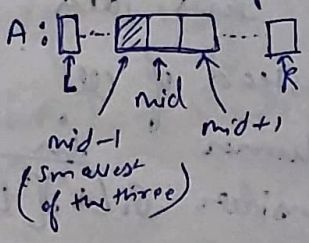
Exercise: ∃ input G on which time $\approx O(n^2)$.
↳ So, = brute-force.

(iii) Let's make this idea more clever, by looking at a 1-D array A[0,...n-1].

Idea: locally-explore using binary-search
@ mid $= \frac{L+R}{2}$ value.



- If A[mid] is not local-minima in A[L....R], then move to a neighboring smaller element: mid-1 or mid+1; move to that half.



▷ $|L_{new} - R_{new}| \leq \frac{1}{2} \cdot |L-R|$

Pf: Since we consider the mid each time.

Left-half is used next; $(R \leftarrow mid-1)$.

- The process ensures the following invariant.
- Lemma 1: $A[L-1] \geq A[L]$; $A[R] \leq A[R+1]$

Pf. If L-1 or R+1 don't exist, then it is vacuously true.

- when they exist, the pseudocode iterates following these inequalities.
- So, we get a proof by induction on the number (#) iterations.

Theorem: This algo. finds local-minima ($A[0 \dots n-1]$) in $O(\log n)$ steps.

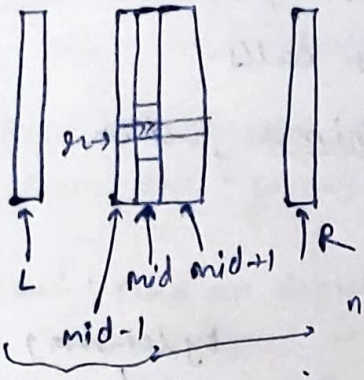
Pf.
- By Lemma 1, there are $\log n$ - steps.
- At the last step: $L = R$, which means $A[L \dots R]$ is a single element x.
- By Lemma 2, $A[L-1] > x < A[R+1]$
  $\Rightarrow x$ is a local-minima

```
Local-min-Array (A[0...n-1]): L=0; R=n-1;
                              found = False;
 while (not found)
   mid ← (L+R)/2;
   if (A[mid] is local-min) found ← true;
   else if (A[mid-1] < A[mid]) R ← mid-1;
   else L ← mid+1; }
 return A[mid];
```

17/01/24

Extend this idea from 1-D to 2-D arrays?

- Instead of cell in an array think of a column in the grid.
- consider $G[*][L \dots R]$; $mid = \frac{L+R}{2}$.
  the Sub-matrix.
- Find the min $G[*][mid] \Rightarrow O(n)$-time.
  $:= \min_{0 \le i < n} G[i][mid]$ (say) $= G[x][mid]$



- Find $P := \min\{G[r][mid-1], G[r][mid], G[r][mid+1]\}$
- ▷ If $P = G[r][mid]$ then output P.
- say, $P = G[r][mid-1]$
- ~~Focus on~~
- Searching in $G[*][L \dots R]$ now.
- ~~else~~
- Formulate an invariant, similar to the 1-D-Lemma 2.

Lemma: $G[*][L-1]$ (just ~~inside~~ outside L) $> G[\min_L][L]$; [$\min_L$ → about L-th column row where min $G[*][L]$ resides]

$G[\min_R][R] < G[*][R+1]$; (just outside R)

[Invariant] This holds at every step of the recursion.

Pf: - Eg. Step as done above that updated R; by moving to the left-half:

- $G[\min_R][R] < G[r][R] < G[r][R+1] < G[*][R+1]$ (mid old)
- 2nd bound. [1st is symmetric]
- This covers the induction-step.
- Ex: Identify the base-case.

⇒ The pf. is finished by induction on # cols.

Lemma: (# steps)
- This algo. halves the number of columns and stops in $O(\log n)$ ~~steps~~ calls.
- $G[\min_L][L]$ is local-minima, when $L=R$.

~~Pf. • $G[*][L-1]$~~

Pf. • $G[*][L-1] > G[\min_L][L] < G[*][R+1]$. (by lemma)

$G[\min_L - 1][L] > G[\min_L][L] < G[\min_L + 1][L]$, (by property of $\min_L$)

$\Rightarrow$ $G[\min_L][L]$ is local-min in the end when $L=R$.

- # recursive calls is $O(\log n)$ as we are (no. of) halving the columns.

Theorem: Local-minima in $G[n][n]$ is found in $O(n \log n)$ time.

Pf. – $O(\log n)$ calls by the previous lemma.
– In each round, it takes $O(n)$ time to find the min in mid-column.

$\Rightarrow \sum_{i=1}^{\log n} O(n) \leq O(n \log n)$

– Further ideas: Reduce $\sum_{i \in \log n} n = O(n \log n)$

to $\sum_{0 \leq i \leq \log n} n/2^i \leq O(n)$

- we can do this by taking the mid-row and picking upper/lower halves!

- Alternate between ~~#columns~~ having the no. of cols and #rows.

- How to argue, what's the invariant?
Q Does the previous algo. ~~idea~~ directly works?

Exercise: Find an input instance, where the column boundary ~~condition~~ invariant, followed by the row bound. invariant doesn't give the right answer.