Another Example: Longest increasing subsequence.

Problem: Given an array $A[1..n]$, we want to know what the length of a longest non-decreasing ~~increasing~~ subsequence is.

Ex: $n = \cancel{6}\ 10$

$A = $ | 4 | 9 | 2 | 7 | 6 | 8 | 1 | 3 | 10 | 5 |

- 4, 9, 10 has length 3
- 2 6 8 10 has length 4
- 2 7 8 10 has length 4
- Any with length 5? No...
- So, the answer is $LIS(A) = 4$

~~Here is an~~

- We can design an algorithm for this problem based upon the following observation:

  Either there is a longest non-decreasing subsequence which uses $A[1]$, or every longest non-decreasing subsequence appears in $A[2..n]$.

- In the first case, we want a non-decreasing subseq ~~elt~~ ~~whose~~ of $A[2..n]$ whose first elt is $\geq A[1]$.

- In the second case, we want a non-decreasing subseq of $A[2..n]$ whose first elt is $\geq -\infty$.

- Try generalizing the problem; given $A[1..n]$ and a number k, find the length of a longest non-decreasing subsequence whose first elt is $\geq k$.

- With this generalization, our algorithm is

<u>LIS(A[1..n], k):</u>

```
if n=0 then return 0
if (A[1] ≥ k) then
    return max{ 1 + LIS(A[2..n], A[1]),
                LIS(A[2..n], k) }
else
    return LIS(A[2..n], k)
```

- The proof that the algorithm is correct is by induction on n.

If $n=0$, the algorithm is correct.

Suppose $n \geq 1$ and consider a longest non-dec. subsequence of $A[1..n]$ whose first elt is $\geq k$; let α be its length.

CASE 1: If this subsequence includes $A[1]$, then $A[1] \geq k$ and ~~α-1~~ $A[2..n]$ contains a ~~stdose~~ non-decreasing subseq. of length $\alpha - 1$ whose first elt is $\geq A[1]$. In fact, the length of ~~the~~ a longest ~~stdos~~ non-decreasing subseq. of $A[2..n]$ whose first elt is $\geq A[1]$ is $\alpha - 1$, or else we could prepend ~~any longer~~ $A[1]$ to any ~~such~~ longer such subseq. and obtain a non-dec. subseq. of $A[1..n]$ whose first elt is $\geq$ ~~k~~ k of length ~~k~~ more than α, a contradiction.

Therefore, by the I.H., $LIS(A[2..n], A[1])$ returns $\alpha - 1$ and our algorithm returns at least α.

Also, the length of a longest non-dec. subseq. of $A[2..n]$ whose first elt. is $\geq k$ is at most α, so by the I.H. $LIS(A[2..n], k)$

returns at most α.

Therefore, in this case, our algorithm returns

$$\max\{1 + LIS(A[2..n], A[1]),\ LIS(A[2..n], k)\}$$

$$= \max\{1 + \alpha - 1,\ \underbrace{LIS(A[2..n], k)}_{\le \alpha}\}$$

$$= \alpha$$

so our algorithm is correct.

CASE 2: The case that the subseq. does not include A[1] is similar and left as an exercise. ■

- The run-time satisfies the recurrence

$$T(n) = 2 \cdot T(n-1) + O(1)$$

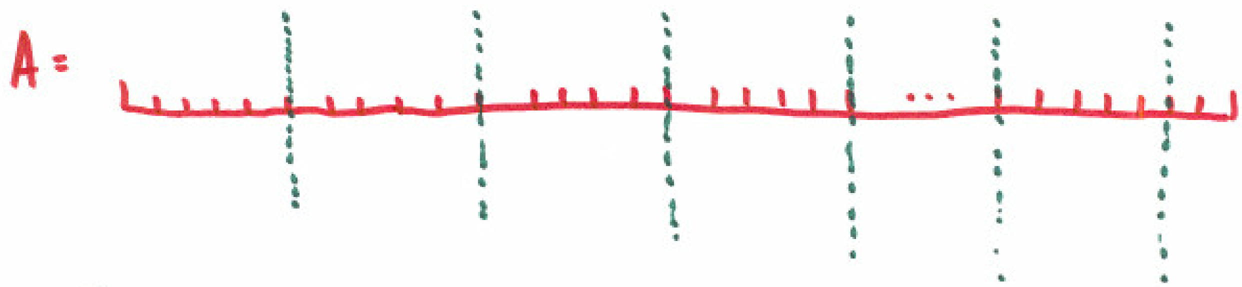
which, using the characteristic eq. method solves to $T(n) = \Theta(2^n)$.

- See CS473 for how to ~~make~~ change this alg to run in polynomial time.

A classic Problem: Select(A[1..n], k)

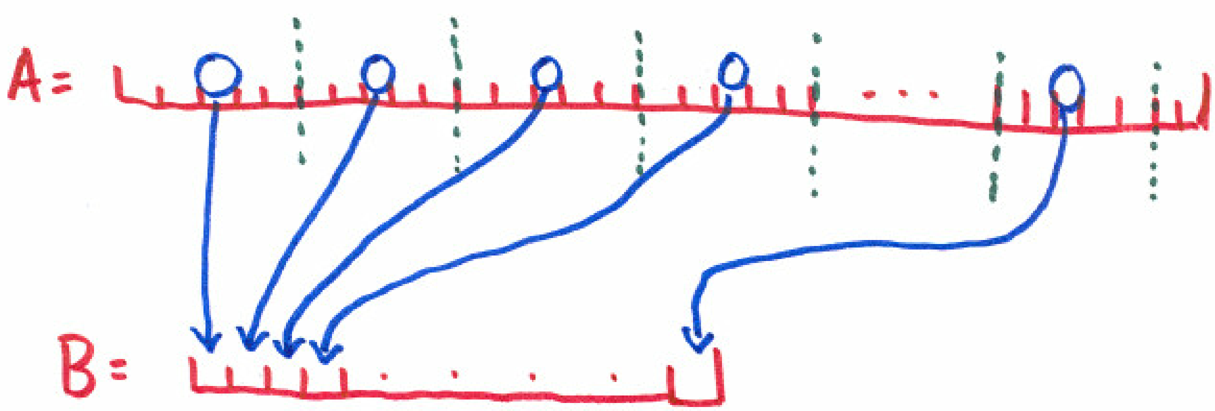
- Given an array $A[1..n]$ of distinct numbers, how quickly can we find the kth smallest element?
- In particular, if $k = \lceil \frac{n}{2} \rceil$ we are asking how quickly can we find the median element?
- If $k = 1$ or $k = n$, we can do this in $O(n)$ time, scanning through the array once.
- For all values of k, we can sort and then lookup for an $O(n \log n)$ algorithm.
- Can we do better?
- Yes; we'll ~~give~~ see an $O(n)$ algorithm.
- Let $T(n)$ be the run-time.

- Here's how. First, split the array $A[1..n]$ into groups of size 5, leaving some extra elts at the end.



- Sort each group of 5 elements. This takes $\lfloor \frac{n}{5} \rfloor \cdot O(1) = O(n)$ time.

- Make a new array B consisting of the median elements from each group of 5.



This takes $O(n)$ time.

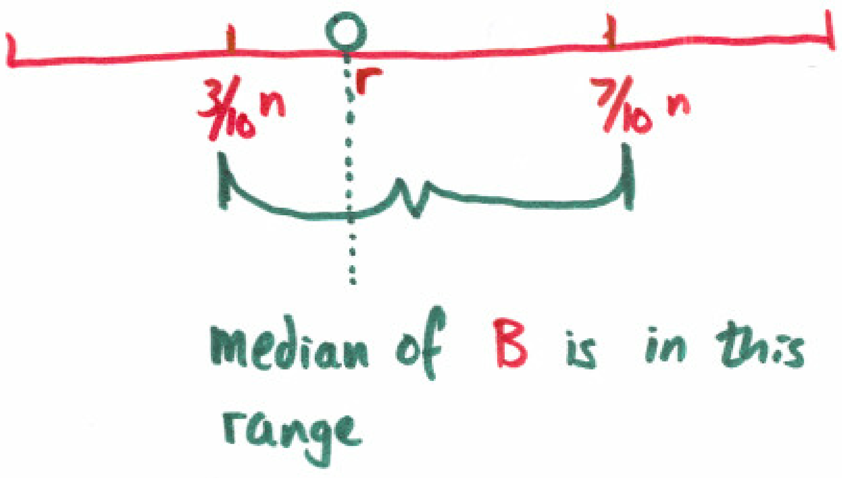
- Recursively, find the median element of B. This takes $T(\frac{n}{5})$ time.
- By scanning through $A[1..n]$, compute the number of elts $\leq$ median of B; this number r is the <u>rank</u> of the median of B in the array A. This takes $O(n)$ time.
- Each group of 5 whose median is $\leq$ median of B cont~~ributes~~ains three elts that are $\leq$ median of B. Therefore there are at least $3\cdot\left(\frac{1}{2}\cdot|B|\right) = 3\cdot\left(\frac{1}{2}\lceil\frac{n}{5}\rceil\right)$

$$\approx \frac{3}{10}n$$

  elements in A that are $\leq$ median of B.
- Similarly, each group of 5 whose median is $\geq$ median of B contains three elts that are $\geq$ median of B, so at least $\frac{3}{10}n$ elements of A are $\geq$ median of B.

- Conclusion: the median of B is in the middle $\frac{6}{10}n = \frac{3}{5}n$ of the elements in A:

sort(A) =



so $\frac{3}{10}n \le r \le \frac{7}{10}n$.

- ~~Next, make two new arrays~~

- If $r = k$, the median of B is the kth smallest element in A and we are done.

- Otherwise, we make two arrays:
  - below[1..r-1]
  - above[1..n-r]

containing all elements in A that are

below ($\leq$) or above ($\geq$) the median of B, respectively. This takes $O(n)$ time.

- If $k < r$, recursively find the kth smallest element in $below[1..r-1]$; this is the kth smallest in A.

  Otherwise, if $k > r$, recursively find the $(k-r)$th smallest element in $above[1..n-r]$; this is the kth smallest in A.

  Because $above$ and $below$ both have at most $\frac{7}{10}n$ elements, this takes at most $T(\frac{7}{10}n)$ time.

- Adding it all up, we get that our run-time is at most

$$
\begin{aligned}
T(n) &\leq T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right) + O(n) \\
&= T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right) + c \cdot n
\end{aligned}
$$

Using the recursion tree method, we see

that ~~each~~ the jth level of the recurrence contributes a total of $c \cdot n \cdot \left(\frac{1}{5} + \frac{7}{10}\right)^j = c \cdot n \cdot \left(\frac{9}{10}\right)^j$ to $T(n)$.

Therefore
$$T(n) \le \sum_{j=0}^{\infty} c \cdot n \cdot \left(\frac{9}{10}\right)^j$$
$$= c \cdot n \sum_{j=0}^{\infty} \left(\frac{9}{10}\right)^j$$
$$= c \cdot n \cdot \frac{1}{1 - \frac{9}{10}}$$
$$= 10 \cdot c \cdot n$$

so $T(n) = O(n)$ and we can find the kth smallest element in linear time.

<u>Remarks</u>: There is nothing special about using groups of 5; any (large enough) constant group size will work.

<u>Exercise</u>: what happens to the run-time if we use groups of size 3? Is 3 large enough?

In practice, this algorithm is not used because the constant $T(n) = \underbrace{10 \cdot c} \cdot n$ is too big.

- An efficient randomized algorithm, known as QuickSelect (analogous to QuickSort) improves on the run-time by choosing an element from A at random and letting that elt. play the role of the median of B; the rest of the algorithm is not changed.

  (Note that it is reasonably likely we'll get lucky and choose an elt. near the middle of A.)

- More about randomized algs in CS473.