- Integer constants count as type int

  Ex: 32 is type int

  - unless specified (in which casting occurs)

  Ex: short x = 32; ⟶ type short

- If you mix & match types, implicit casting occurs

  int x = ____;

  size_t y = ____;

  y = x; // Implicit casting (y = (size_t)x;)

  if(x < y); // Implicit casting! Don't worry about what happens, though. Always be explicit

Ex: Assume short 2 bytes, int 4 bytes,

Soln:

short a = 0xDD; // What's the value? value: 235

signed char a0 = (signed char)a; Impl. defined!

unsigned char a1 = (unsigned char)a; hex: 0xDD val: 221

short b = -3; hex: 0xFFFD

int b0 = (int)b; hex: 0xFFFF FFFD val: -3

unsigned char b1 = (unsigned char)b; hex: 0xFD val: 253

size_t y = -1; hex: 0xFFFF FFFF FFFF FFFF val: 1.84×10^{19}

☆ What happens? If not implementation-defined, what is value? what is hex rep?

LI Proofs

- LI's used to prove correctness & safety
  - LI's used to prove correctness & safety
  - Safety - proving ONLY preconditions are satisfied
- LI's are always checked before LG, even if LG is false

Prec
↓
LI ← Loop
↓
LG
↓
Post

How to Prove LI's

- Init: Show LI's are true before 1^{st} iteration
  - Can use prec, variable initializations, and vacuous truths (empty array is sorted)
- Pres: Assume: paste in LI (at arbitrary iteration)
  To Show: paste in LI, but with primed variables (for ones that change) (is true at end of iter)
  - Can use LG, LI, & loop code
- Term: Uses operational reasoning
  - "The quantity [expression] strictly (incr / decr) until it reaches upper/lower bound based on LI's upon which LG is false and exit loop"
- Exit: After loop exits, prove postconditions
  - Can use LG (negation), and LI's (still true)

☆ Tips:
- Put in LG & the Assumes! (for pres)
- From there, loop code should directly get you to-shows
  - plug in primed variables!

Searching & Sorting

- Linear search, binary search, selection sort, quicksort, mergesort
- Don't need exact implementation, but just know how they work and compute big-O bounds
  - apply previous knowledge to new setting

- Stable sorting

  2 1 3 5 4 5

  1 2 3 4 5 5 ← Stable (only swap elems with STRICT inequality)

  1 2 3 4 5 5 ← unstable

- In-place Sorting:
  - ↳ uses O(1) extra space
  - ↳ only uses a couple of variables, rather than whole arrays

	Linear Search	Binary Search ↳ requires SORTED array
Best Case	$O(1)$	$O(1)$
Worst Case	$O(n)$	$O(\log n)$

	Selection	Quick	Merge
Best Case	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Worst Case	$O(n^2)$	$O(n^2)$	$O(n \log n)$
Stable	NO	NO	YES
In-place	YES	YES	NO

Linear Search

0	1	2	3	4	5	6

- Go from beginning to end until find elem

Binary Search

$x = 5$

0	1	2	3	4	5	6
0	3	5	9	10	15	17

lo hi

lo mid hi

lo hi

lo: 0
hi: 6
mid: 3

lo: 0
hi: 3
mid: 1

lo: 1
hi: 3
mid: 2
Found it!

- Search mid = lo + (hi - lo) / 2
  - don't do (lo + hi) / 2
    it might overflow
- if A[mid] == x, return mid
- if A[mid] < x, search right of mid
  (set lo = mid)
- if A[mid] > x, search left of mid
  (set hi = mid)

Selection Sort

0	1	2	3	4	5	6
9	2	10	6	3	5	1

i = 0

0	1	2	3	4	5	6
1	2	10	6	3	5	9

i = 1

0	1	2	3	4	5	6
1	2	10	6	3	5	9

i = 2

Work: $n + n-1 + \dots + 2 + 1$

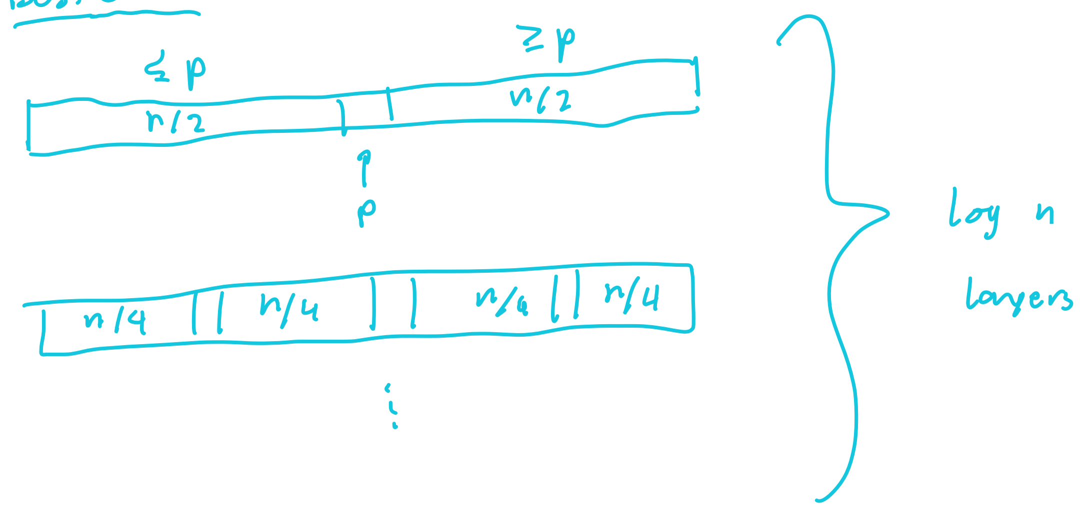
$= \frac{n(n+1)}{2} = O(n^2)$

```
i = 0
while i < n:
    Find min elem in A[i, n)
    Swap with A[i]
    i++;
```

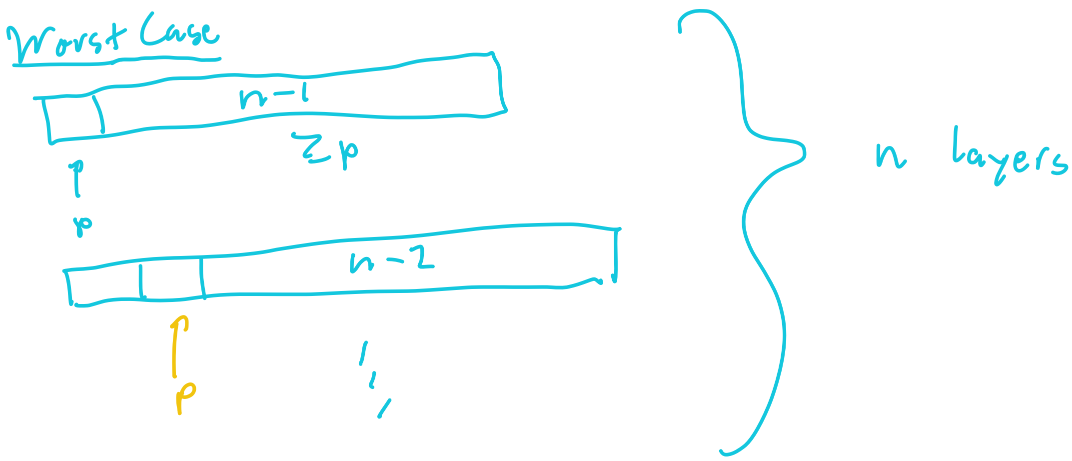
Quick Sort

- Pick pivot & move elements left or right
- Repeat on both halves

Best Case



Worst Case



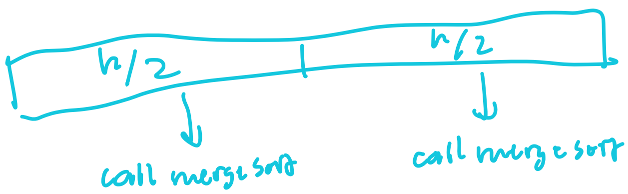
★ Each layer is $O(n)$ since need to compare everything with pivot

Worst: $O(n^2)$

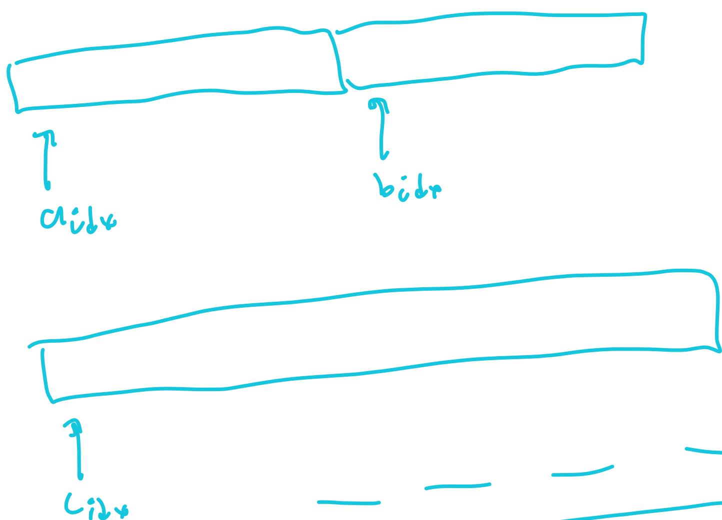
Best: $O(n \log n)$

Merge Sort

- Recursively call mergesort on each half
- merge both halves

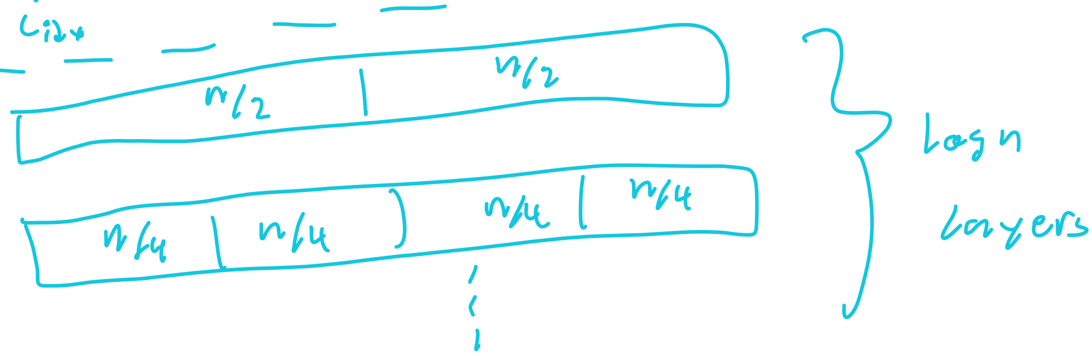


- Each half now sorted
- merge easy!



- put in array element that is smaller A[aidx] A[bidx]
- then increment one of them
- incr. cidx

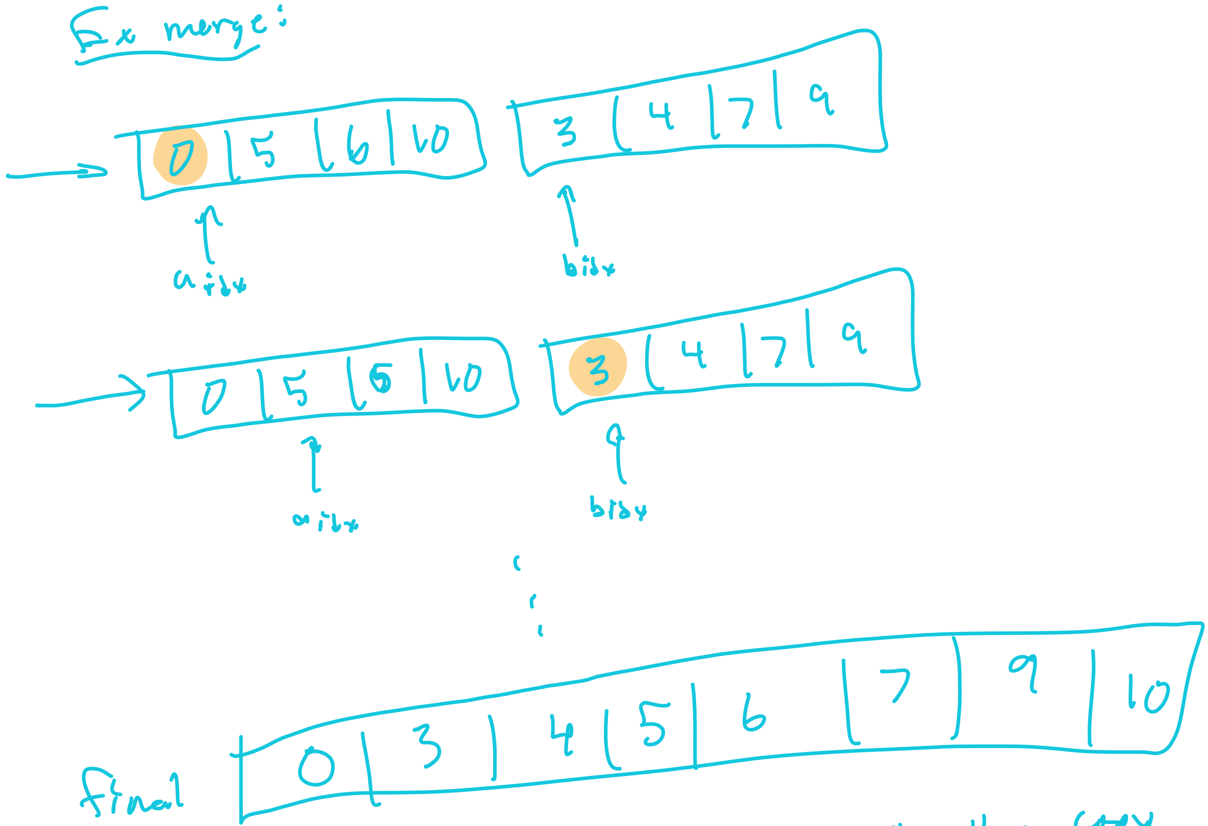
copy of array where you slowly created sorted array



$O(n \log n)$

- Each layer is $O(n)$ for merging
  - log n layers

Ex merge:



- If finished w/ one array, but not the other, copy unfinished parts over