* # comparison in quick sort

$$C_{avg}(n) = n-1 + \frac{1}{n}\sum_{i=1}^{n}\left(C_{avg}(i-1) + C_{avg}(n-i)\right)$$

$$= (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} C_{avg}(i)$$

$$-) \; C_{avg}(n-1) = (n-2) + \frac{2}{n-1}\sum_{i=0}^{n-2} C_{avg}(i)$$

$$n C_{avg}(n) - (n-1) C_{avg}(n-1) = n(n-1) - (n-1)(n-2) + 2 C_{avg}(n-1)$$

$$= 2n-2 + 2 C_{avg}(n-1)$$

$$\Rightarrow n C_{avg}(n) - (n+1) C_{avg}(n-1) = 2n-2$$

$$\frac{C_{avg}(n)}{n+1} = \frac{C_{avg}(n-1)}{n} + \boxed{\frac{2n-2}{n(n+1)}}$$

$$\boxed{\frac{2}{n} - \frac{4}{n(n+1)}}$$

$$\frac{C_{avg}(n)}{n+1} = \frac{C_{avg}(0)}{1} + \sum_{i=1}^{n}\left(\frac{2}{i} - \frac{4}{i(i+1)}\right)$$

$$= 2\boxed{\sum_{i=1}^{n}\frac{1}{i}} - 4\boxed{\sum_{i=1}^{n}\left(\frac{1}{i} - \frac{1}{i+1}\right)}$$

$$\boxed{O(\log n)} \qquad \boxed{-\frac{1}{n+1}}$$

$$\Rightarrow C_{avg}(n) = O(n \log n)$$

* comparison sort

selection, tournament, merge, heap,
bubble, insertion, shell, quick

needs $\Omega(n \log n)$ comparisons

* consider a tree like the balance-coin
leaf: a permutation of the original array
node: comparisons

leaves $\geq$ # permutations to make sorting correct

$\Rightarrow$ # leaves $\geq n!$
$\Rightarrow$ depth $\geq \log_2(n!) + 1 = \Omega(n \log n)$
$\Rightarrow$ need $\Omega(n \log n)$ comparisons

* non-comparison sort

counting sort for keys in $\{1, 2, \cdots, K\}$

① construct an array c of size K ($O(K)$ space, $O(1)$ time)
② count (# key = k) and store in c[k] ($O(n)$ time)
③ compute start[k] = $\sum_{i=1}^{k-1} c[i]$ ($O(K)$ space, $O(K)$ time)
④ put element a[i] in pos. start[k] + ○ ($O(n)$ time)

recall: sparse matrix transpose $\approx$ counting sort on column index

$O(n+K)$ time, $O(K)$ space

* sorting integers

179 , 208 , 306 , 93 , 859 , 984 , 55 , 9 , 271 , 33

① compare by hundreds (say, count sort)

984
859
306
208 , 271
179
093 , 055 , 009 , 033

② count sort sublist by tens

208 271
009 033 055 093

③ final result

984 859 306 208 271 179 009 033 055 093

called most-significant digit (MSD) sort

* least-significant first ?

271		306		009	
093		208		033	
033		009		055	
984	⇒	033	⇒	093	
055	stable	055	stable	179	LSD-sort
306		859		208	time { O(n+r) per pass, d passes
208		271		271	
179		179		306	
859		984		859	space { O(r) for counting sort, O(n) for temp storage array
009		093		984	
↳ sorted		↳ sorted		↳ sorted	

* on integers of base r : radix-r LSD/MSD
on other key types : LSD/MSD

* bin / bucket sort :
  similar to counting sort but maintain elements in each bin (rather than count & put)

* quick sort : practically best "avg performace" for large data
  merge sort : " "worst " "
  insertion sort : for small data

  can be mixed !

* external sort :
  not teaching, mostly extended merge / tournament sort

* want

a[key] = value (insert) and x = a[key] (retrieve) fast for some keys w/ order

ordered array (like sparse matrix)

O(n) insert O(log n) retrieve

linked list w/o order

O(1) insert O(n) retrieve

ordered skip list

O(log n) insert O(log n) retrieve

"good" BST

O(log n) insert O(log n) retrieve

"bad" BST

O(n) insert O(n) retrieve

* if keys are integers

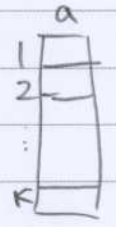
O(1) insert O(1) retrieve

by primitive array

how about using a "simple" function h: key → integer
O(1)

a[h(key)] = value

x = a[h(key)]



if {key} ⟹ {1, 2, ···, K} , yes!
one-to-one
perfect hashing

* what if not ?

keys more than K ⇒ two keys of same h(·) outcome

synonyms

⇒ collision

e.g. h(str) = str[0] - 'a';

h("act") = h("addition")

	a
0	act
1	banana
2	
3	dog
4	

addition, as, am

* how to solve collision during insertion?
  1. fixed (unordered) array per position : can still overflow
  2. linked list per position : chaining
  3. other data structure per position : more complicated but doable
  4. use other empty positions : probing

* Chaining :
  a[key] = value ⇒ insert (key, value) to chain h(key)
  x = a[key] ⇒ linear (sequential) search chain h(key) for (key, value)

* probing (open addressing)
  a[key] = value ⇒
  1. insert (key, value) to spot $h_0(key) = h(key)$
  2. if fail, insert (key, value) to spot $h_1(key)$
  3. if fail, .. $h_2(key)$
  ⋮
  m-1. .. $h_m(key)$
  m. declare failure

  x = a[key] ⇒
  1. check spot $h_0(key) = h(key)$
  2. if not found, check spot $h_1(key)$
  ⋮
  m-1. $h_m(key)$
  m. declare not found

variants:
- (a) m different hash functions h_m : rehashing
- (b) $h_i(key) = (h_{i-1}(key) + 1) \% \underbrace{b}_{\#\text{ spots}}$: linear probing
- (c) $h_i(key) = (h_{i-1}(key) + rand) \% b$: random probing
- (d) $h_i(key) = \begin{cases} (h_0(key) + i^2) \% b \\ (h_0(key) - i^2) \end{cases}$: quadratic probing