Algorithms

- Algorithm: sequence of well-defined computational procedures that takes some value(s) as input and produces some value(s) as output

- Algorithm as a tool for solving a well-specified computational problem
- Any input, meeting the conditions of the problem statement is called an instance of the problem, which is later used to compute a problem
- An algorithm must be correct, by correct it means that the algorithm solves a computational problem
- Choosing an efficient algorithm is as important as choosing a fast hardware

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$	0	5.9	11.8				
$\sqrt{n}$	1	7.79	60				
n	1	60	3600				
$n \lg n$	0	354					
n^2	1	3600					
n^3	1	21600					
2^n	1	1.15×10^{19}					
$n!$	1	8.32×10^{81}					

Insertion Sort

- In-place Algorithm

→ Algo that does not use extra space and produces an output in the same memory that contains data, can use constant extra space for variables

- INSERTION SORT(A):

```
for j ← 2 to A.length
    key ← A[j]
    i ← j-1

    while i > 0 and A[i] > key
        A[i+1] ← A[i]
        i ← i-1
    A[i+1] ← key
```

- situations to use in
- Already sorted / Nearly sorted data
- Small Datasets
  → simple
  → low constant factors

- Systems with minimal memory usage
  → space complexity of O(1)

- online sorting
  → sequential data

- Worst-Case : $O(n^2)$
- Average-Case: $\Theta(n^2)$
- Best-case : $\Omega(n)$
- In-place Algo
- Space Complexity: O(1)

- RAM Model (Random Access Machine)
- instructions executed one after another, no concurrent execution
- An abstraction, which allows us to ignore hardware specifications, and purely focuses on algorithmic logic only

- To measure efficiency as a "standard":
- Each step takes 1 unit of time in RAM model
- constants are dropped, and only order of growth is measured
  → Machine differences : a faster computer might execute more steps in the time a slower machine executes 1.
  → compiler differences: Different languages require different number of machine instructions for the same line of code

- Asymptotic Analysis
  → standard measurement focuses on performance change as input size approaches infinity

Q- Apply RAM Model and calculate Running Time (Time)

```
x ← 0                                          c1
for i ← 1 to n                                 c2
    temp ← i+1                                 c3
    for j ← 1 to n                             c4
        If ( j mod 2 = 0) Then                 c5
          x ← x + temp                         c6
        Else                                   c7
          x ← x - 1                            c8
```

Individual Cost	Repetition	Total
c_1	1	c_1
c_2	$n+1$	$c_2(n+1)$
c_3	n	$c_3(n)$
c_4	$n(n+1)$	$c_4(n^2+n)$
c_5	n^2	$c_5(n^2)$
c_6	$n^2(k)$ $0 \le k \le 1$	$c_6(n^2k)$
c_7	$n^2(1-k)$	$c_7[n^2(1-k)]$
c_8	$n^2(1-k)$ $0 \le k \le 1$	$c_8[n^2(1-k)]$

$$T(n) = c_1 + c_2(n+1) + c_3(n) + c_4(n^2+n) + c_5(n^2) + c_6(n^2k) + c_7[n^2(1-k)] + c_8[n^2(1-k)]$$

$$= n^2(c_4 + c_5 + kc_6 + c_7 - kc_7 + c_8 - kc_8) + n(c_2 + c_3 + c_4) + (c_1 + c_2)$$

$$= n^2(c_4 + c_5 + c_7 + c_8 + k(c_6 - c_7 - c_8)) + n(c_2 + c_3 + c_4) + (c_1 + c_2)$$

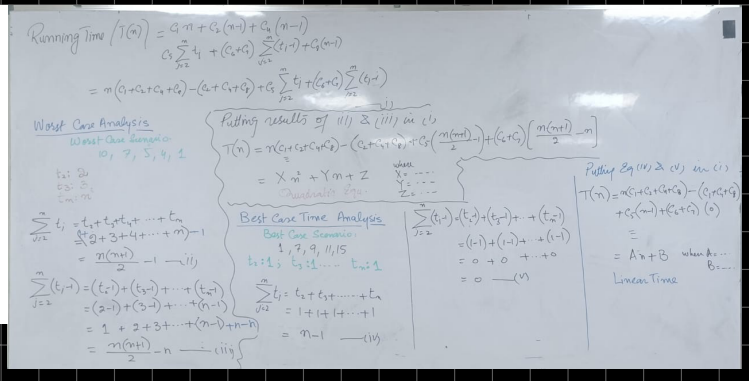
$T(n) = An^2 + Bn + C$, a quadratic-time algorithm

where $A = c_4 + c_5 + c_7 + c_8 + k(c_6 - c_7 - c_8)$

$B = c_2 + c_3 + c_4$

$C = c_1 + c_2$

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j - 1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$



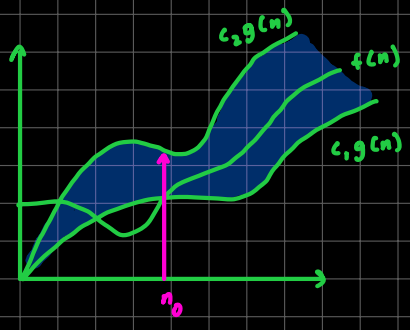
Growth of Functions

- Asymptotic notation for expressing algorithm's running time.
- Applied on functions
- Can be used to represent the amount of space algorithm use

- $\Theta(g(n))$

→ $\Theta(g(n)) = \{ f(n)$: there exist +ve constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

- It means a function exists that belongs to the set $O(g(n))$ if there exist +ve constants c_1 and c_2 such that it can be sandwiched b/w $c_1 g(n)$ and $c_2 g(n)$ for large n

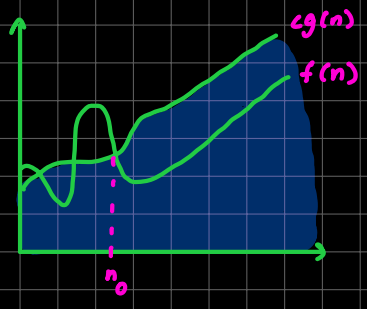


- $f(n) = \Theta(g(n)) \approx f(n) \in \Theta(g(n))$] same case for all other notations

- Asymptotically tight bound
- Algo grows at a precise rate, sandwiched average case where best and worst case growth rates match

- $O(g(n))$

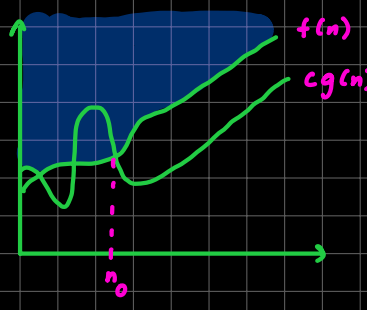
- $O(g(n)) = \{ f(n)$: there exist +ve constants c and n_0 such that $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0 \}$



- Gives asymptotic upper bound
- Algo will not grow any faster than the rate, ceiling on growth

- $\Omega(g(n))$

- $\Omega(g(n)) = \{ f(n)$: there exist +ve constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$



- if $f(n) = O(g(n))$ and $f(n) = \Theta(g(n))$, only then $f(n) = \Omega(g(n))$

- Algorithm will not grow slower than this rate, puts a floor on its growth

Divide & Conquer Approach

- Aim is to divide a problem into sub-problems.
- Solve the sub-problems which are the same instances of the original problem but smaller in size
- conquer the sub-problems by solving them recursively
- Combine the solutions of the subproblem into the solution for the original problem

a- Merge - sort

- MergeSort (A, s, e)

  If (s < e) —— c_1 — 1] divide $\Theta(1)$

  $m \leftarrow \lfloor \frac{s+e}{2} \rfloor$ —— c_2 — 1

  MergeSort (A, s, m) — $T(\frac{n}{2})$] conquer

  MergeSort (A, m+1, e) — $T(\frac{n}{2})$

  Merge (A, s, m, e) — $\Theta(n)$] combine

$T(n) = c_1 + c_2 + T(\frac{n}{2}) + T(\frac{n}{2}) + \Theta(n)$

$= 2T(\frac{n}{2}) + \Theta(n)$ => Recurrence Eq

$T(n) = aT(\frac{n}{b}) + D(n) + C(n)$ — Generic eq

a = N.o of sub-problems

b = size of subproblems that are being divided

- $D(n) = \Theta(1)$ → Divide step
- $2T(\frac{n}{2})$ → conquer step
- $\Theta(n)$ → Combine step

- Note: Recursion "bottoms out" when sequence to be sorted has length 1 is already in sorted order

- Merge (A, p, q, r)

  $n_1 \leftarrow q - p + 1$

  $n_2 \leftarrow r - q$

  Let $L[1 \cdots n_1+1]$, $R[1 \cdots n_2+1]$ be new Arrays

  for $i \leftarrow 1$ to n_1

    $L[i] \leftarrow A[p+i-1]$

  for $j \leftarrow 1$ to n_2

    $R[j] \leftarrow A[q+j]$

  $L[n_1+1] \leftarrow \infty$

  $R_2[n_2+1] \leftarrow \infty$

  $i \leftarrow 1$

  $j \leftarrow 1$

  for $k \leftarrow p$ to r

    if $L[i] \leq R[j]$

      $A[k] \leftarrow L[i]$

      $i \leftarrow i+1$

    else

      $A[k] \leftarrow R[j]$

      $j \leftarrow j+1$

- Worst-case = Best-case = Average-case = $O(n \log n)$
- Space Complexity = $O(n)$

- Cases to be used in:
- Large Datasets

→ complexity remains same at any dataset, other algos degrade in performance

- Sorting linked lists
- Parallel processing
- Inversion count problems

- External sorting

→ when data is large to fit into main memory

- Stable-sorting requirement

→ Preserves original order of identical values; databases

Solving Recurrences

① Substitution Method

① Guess the Form
② verify by induction
③ solve for constants

Q- $2T(\frac{n}{2}) + n \longrightarrow 2+2=4$

1- Take a guess for $O(g(n))$

$O(n\lg n)$, so $T(n) \leq cn\lg n$ - introduce a constant

$T(\frac{n}{2}) \leq c(\frac{n}{2})\lg(\frac{n}{2})$

$T(n) \leq 2c(\frac{n}{2})\lg(\frac{n}{2}) + n$ - Sub in the original eq

$\leq cn[\lg n - \lg 2] + n$

$\leq cn\lg n - cn + n$

$\leq cn\lg n - (cn - n)$

For only $T(n) \leq cn\lg n$, leftover part must be 0.

$T(n) \leq$ guess for $O(g(n))$ // $n - cn \leq 0$
$T(n) \geq$ guess for $\Omega(g(n))$ // $n - cn \geq 0$
· satisfy above both for $\Theta(g(n))$

$-cn + n \leq 0$
$n \leq cn$
$1 \leq c$
$c \geq 1$

→ Check hypothesis

$T(n) = 1$ // Assumption

$1 \leq c1\lg 1$
$1 \leq 0$ → False

$c=2, T(2)=4$

$4 \leq 2 \cdot 2\lg 2$
$4 \leq 4$ True

② Iterative Method

Q- $T(n) = 2T(\frac{n}{2}) + n$

$T(n) = 2[2T(\frac{n}{4}) + \frac{n}{2}] + n$

$= 4T(\frac{n}{4}) + n + n$

$= 4T(\frac{n}{4}) + 2n$

$T(n) = 4(2T(\frac{n}{8}) + \frac{n}{4}) + 2n$

$= 8T(\frac{n}{8}) + 3n$

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + \frac{n}{2}$

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + \frac{n}{4}$

· $2^k T(\frac{n}{2^k}) + kn$

· To hit the base-case

$\frac{n}{2^k} = 1 \Rightarrow n = 2^k$
$k = \lg n$

- $2^k T\left(\frac{n}{2^k}\right) + kn$

→ $2^{\lg n} T(1) + \lg n(n)$

→ $n(1) + n\lg n$ // Assuming $T(1)=1$, $2^{\log_2 n} = n$

$n + n\lg n$

⇒ $O(n\lg n)$ // Same for Θ, Ω

$=$

③ Masters Theorem

- Works for recurrences:

$aT\left(\frac{n}{b}\right) + f(n)$

a = # of sub-problems
b = factor by which size input shrinks
f(n) = cost to divide and combine the work

- Case 1:

$f(n) = O(n^{\log_b a - \varepsilon})$, $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

- Case 2:

$f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

- Case 3:

$f(n) = \Omega(n^{\log_b a + \varepsilon})$ $\varepsilon > 0$

→ $af\left(\frac{n}{b}\right) \leq cf(n)$ ⇒ $T(n) = \Theta(f(n))$
$c < 1$, and sufficiently large n

Q- $T(n) = 4T\left(\frac{n}{2}\right) + n$

$a=4$, $b=2$, $f(n) = n$

$f(n) = O(n^{\lg 4 - \varepsilon})$
$= O(n^{2-\varepsilon})$ $\varepsilon = 1$
$n = O(n)$ ✓

$n = \Theta(n^{\lg 4})$
$n = \Theta(n^2)$
$=$

Q- $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

→ Case 1 fails so we move to case 2

$a=4$, $b=2$, $f(n) = n^2$

$f(n) = O(n^{\lg 4 - \varepsilon})$
$= O(n^{2-\varepsilon})$ no value
$=$

x ——————————— x

$f(n) = \Theta(n^{\lg 4} \lg^k n)$
$n^2 = \Theta(n^2 \cdot (\lg n)^k)$
$n^2 = \Theta(n^2)$ ✓ $k=0$

$n^2 = \Theta(n^{\lg 4} \cdot (\lg n)^1)$

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

Q- $2T\left(\frac{n}{2}\right) + n$

$a=2$, $b=2$, $f(n) = n$

$f(n) = \Theta(n^{\lg 2} \cdot \lg^k n)$
$n = \Theta(n)$ $k=0$

$n = \Theta(n \cdot \lg n)$

Q- $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

$a=4,\ b=2,\ f(n)=n^2$

$n^2 = \Omega(n^{\log_2 + \varepsilon})$

$n^2 = \Omega\left(n^{1+1}\right)\ \varepsilon = 1$

$n^2 = \Omega(n^2)$ ✓

$2f\left(\frac{n}{2}\right) \leq cf(n)$

$2 * \frac{n^2}{4} \leq cn^2$ ✓

$\frac{n^2}{2} \leq \frac{1}{2}n^2$ ✓

Q- $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

$a=4,\ b=2,\ f(n)=n^3$

$n^3 = \Omega(n^{\log 4 + \varepsilon})$

$n^3 = \Omega(n^{2+\varepsilon})$

$n^3 = \Omega(n^3)$ ✓ $\varepsilon = 1$

$4f\left(\frac{n}{2}\right) \leq cf(n)$

$4\left(\frac{n}{2}\right)^3 \leq cn^3$

$4 * \frac{n^3}{8} \leq \frac{1}{2}n^3$ ✓

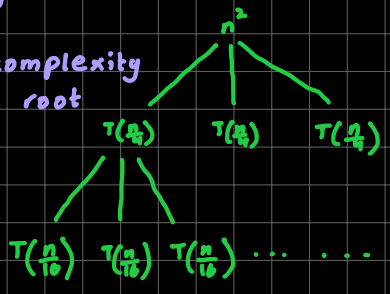
$n^3 \leq n^3$ ✓

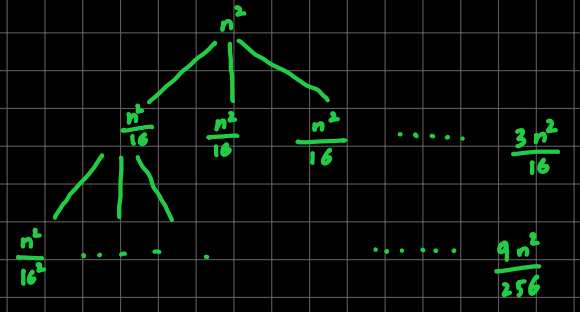
(4) Recurrence Tree

Q- $T(n) = 3T\left(\frac{n}{4}\right) + n^2$

- Decreasing geometric series
- $r<1$, total complexity dominated by root

① Make pattern
② Calculate height of tree
③ Total summation





Level $0 = n^2$

" $1 = \frac{3}{16}n^2$

" $2 = \frac{3^2}{16^2}n^2$

Level $i = \frac{3^i}{16^i}n^2$

→ Decreasing geometric sequence b/c $r<1$ ∴ $\frac{3}{16}$

→ Height, since tree is being divided by a factor of 4

$\frac{n}{4^h} = 1$ $\quad 4^h = n$

$h = \log_4 n$

→ Summation

$T(n) = \sum_{0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i n^2 +$ cost of leaves

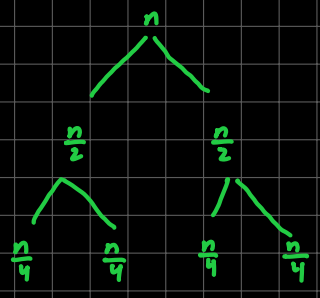
$T(n) \approx n^2 \times \frac{1}{1-\frac{3}{16}}$

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

Q- $T(n) = 2T(\frac{n}{2}) + n$

· Increasing Costs

· r=1, total complexity dominated by every level



--- $2 \times \frac{n}{2} = n$

.... $4 \times \frac{n}{4} = n$

→ Total Cost= Cost of each level x number of levels

→ height of tree

$\frac{n}{2^k} = 1$

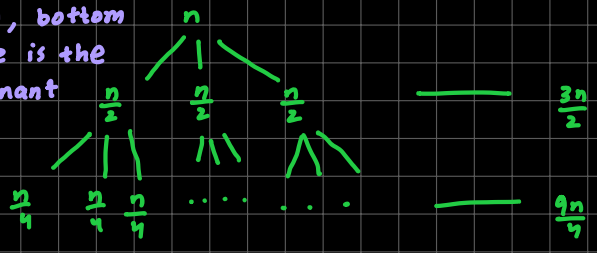
$2^k = n$

$k = \log_2 n$

→ Cost= $n \times \log_2 n$ → $T(n) = \Theta(n \log n)$

Q- $3T(\frac{n}{2}) + n$

· r>1, bottom value is the dominant



$(\frac{3}{2})^i n$

$n + \frac{3}{2}n + \frac{9}{4}n + \frac{27n}{8} \cdots$

① height → $\frac{n}{2^h} = 1$

$2^h = n$

$h = \log_2 n$

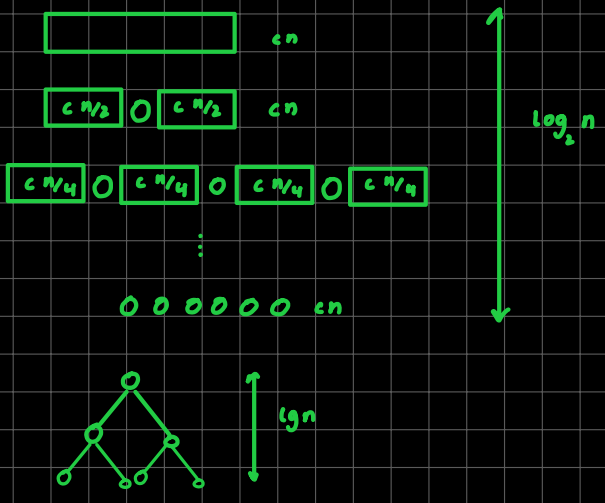
② number of leaves = $3^{\text{height}} = 3^{\log_2 3}$

$T(n) = \Theta(n^{\log_2 3})$

Quicksort

```
Quicksort (A,s,e)
    if (s<e)
        m ← Partition(A,s,e)   — Θ(n)
        Quicksort(A,s,m-1)
        Quicksort(A,m+1,e)
```

· Best-Case Analysis



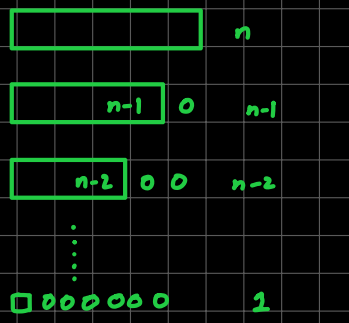
$T(n) = 2T(\frac{n}{2}) + \Theta(n)$

$= cn.\log n$

$= \Theta(n\log n)$

$= \Omega(n\log n)$

· Worst-Case Analysis



$T(n) = n + (n-1) + \dots + 1()$

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

$= \Theta(n^2)$

$= O(n^2)$

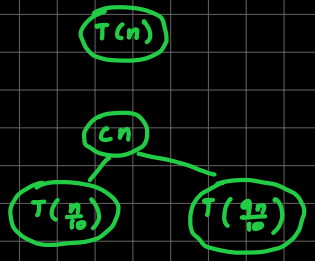
· Average-Case Analysis

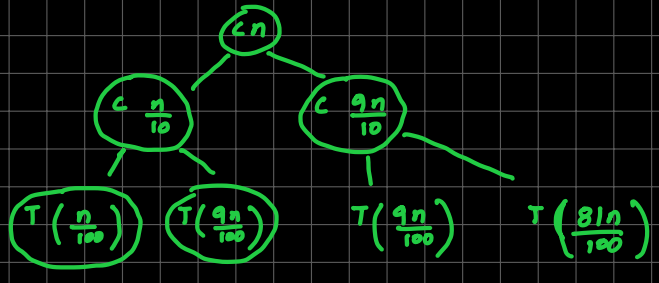
10:90

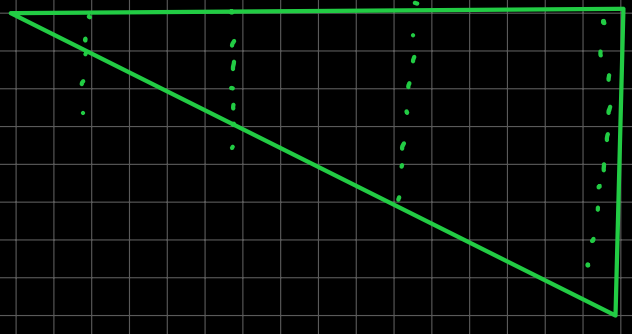
$T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + \Theta(n)$

$T(\frac{n}{10}) = T(\frac{n}{100}) + T(\frac{9n}{100}) + \Theta(\frac{n}{10})$

$T(\frac{9n}{10}) = T(\frac{9n}{100}) + T(\frac{81n}{100}) + \Theta(\frac{9n}{10})$







· Tree
# of nodes: n
height: log n
leaves: $\frac{n}{2}$ approx

· Array
· elements: n
· Permutation: n!
· Branches: n!
· Leaves: n!
· Total nodes: 2n!
· height: $\log(2n!) \approx O(n\log n)$

- Suitable
- In-place → used in limited memory situations
- Finding k^{th} smallest / largest element in an unsorted array

Count Sort

- CountSort (A,B,k)

  // let C[0...k] be a new array

  for i ← 0 to k
    C[i] ← 0

  for j ← 1 to A.length
    C[A[j]] ← C[A[j]] + 1

  // C[i] now contains the number of elements equal to i

  for i ← 1 to k
    C[i] ← C[i] + C[i-1]

  for j ← A.length to 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1

$\Theta(n)$
used when $k = O(n)$

- stable