Algorithms. → How we solve algorithm efficiently.

Time Complexity → How runtime scales?
(growth of running time w.r.t. input size).

- Depends on input size.
  obtain formula for function $T(n)$ to capture growth of runing time wrt input size
- Access basic operation take constant time.
- Ignores constants and lower order terms.
  (focus on dominating terms).

ex) $T(n) = c_1 + \sum_{i=1}^{n} c_2 + c_3 = c_2 n + c_1 + c_3 = \theta(n)$.

Asymptotic time Complexity

$\theta(\cdot)$

Nested looop:
if inner loop depends on outer loop, complexity is normally $\theta(n^2)$.

```
for i in range(1, n):
    for j in range(i, n):
```

$$\underset{\substack{\uparrow \\ \text{for } k \text{ outer iteration.}}}{(n-1)+(n-2)+\cdots+(n-k)+\cdots+(n-n)}$$

$= 1+2+3+\cdots+(n-1)$ ← $1+2+3+\cdots+m = \dfrac{m(m+1)}{2}$

$= \dfrac{n(n-1)}{2} = \binom{n}{2}$

$= \theta(n^2)$

Tips

① Find formula $f(k)$ for "number of iterations of inner loop during outer iteration k"

② Then sum up total cost $\sum_{k=1}^{n} f(k)$.

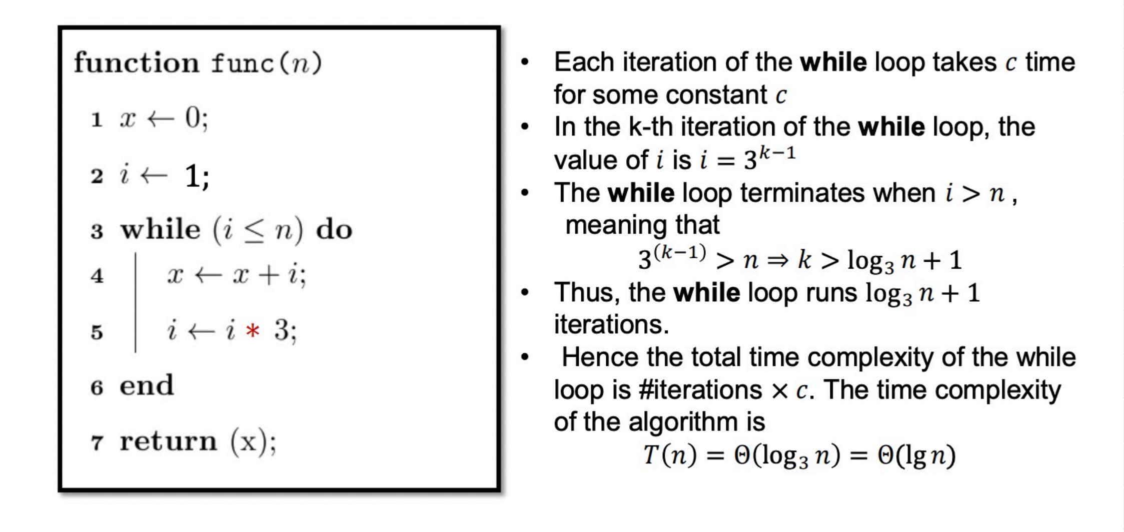
Common growth rate

- $\theta(1)$
- $\theta(\log n)$
- $\theta((\log n)^2)$
- $\theta(n^{0.5})$
- $\theta(n)$
- $\theta(n\log n)$
- $\theta(n^2)$
- $\theta(n^3)$
- $\theta(2^n)$
- $\theta(n^n)$

Solution Formulation:

- Analyze the inner loop, keep track of iteration value.
- Determine the number of iteration the loop will run
- Calculate complexity, capture dominate term.

  $T(n)$



```
function func(n)
  x ← 0;
  i ← 1;
  while (i ≤ n) do
      x ← x + i;
      i ← i * 3;
  end
  return (x);
```

- Each iteration of the **while** loop takes c time for some constant c
- In the k-th iteration of the **while** loop, the value of i is $i = 3^{k-1}$
- The **while** loop terminates when $i > n$, meaning that
  $$3^{(k-1)} > n \Rightarrow k > \log_3 n + 1$$
- Thus, the **while** loop runs $\log_3 n + 1$ iterations.
- Hence the total time complexity of the while loop is #iterations × c. The time complexity of the algorithm is
  $$T(n) = \Theta(\log_3 n) = \Theta(\lg n)$$

Asymptotic Notation.

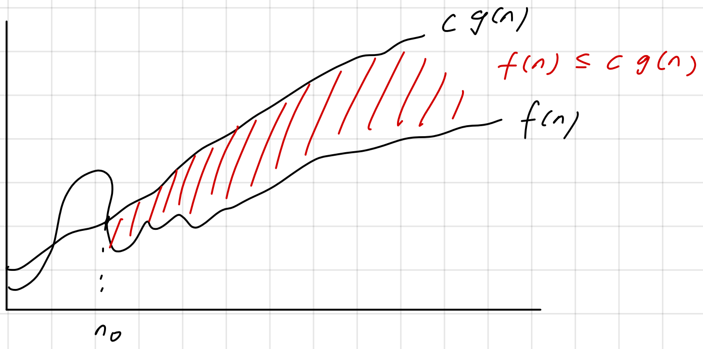
- Big-O notation

Def: We write $f(n) = O(g(n))$ if there are positive constant n_0 and c such that for all $n \geq n_0$: $f(n) \leq c \cdot g(n)$.

$f(n) \in O(g(n))$

$f(n) = O(g(n))$ means that

- $f(n)$ grows at fast as $g(n)$
- $g(n)$ is an asymptotic upper bound for $f(n)$.



ex) ...

Lemma [upper bound].

- If $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists, then $f(n) = O(g(n)) \iff$ (tight upper bound)

  $\lim_{n \to \infty} \frac{f(n)}{g(n)} \le c$, where c is a positive constant.

Corollary [upper bound].

- If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$, then $f(n) = O(g(n))$
- if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = +\infty$, then $f(n) = O(g(n))$ does not hold.

- Big-Ω notation

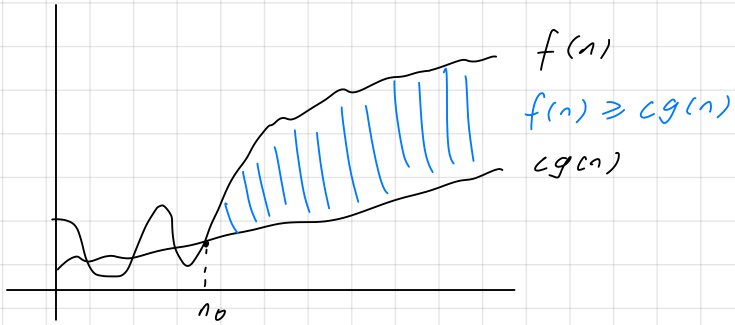
Def: $f(n) = \Omega(g(n))$ if there are positive constant n_0 and c such that for all $n > n_0$:

$$f(n) \geq c \cdot g(n).$$

$f(n) \in \Omega(g(n))$.

$f(n)$ grows at least as fast as $g(n)$

$g(n)$ is an asymptotic lower bound.



Lemma [lower bound].

- If $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists, then $f(n) = \Omega(g(n)) \iff$ (tight lower bound)

  $\lim_{n \to \infty} \frac{f(n)}{g(n)} \geq c$, where c is a positive constant.

Corollary [lower bound].

- If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = +\infty$, then $f(n) = \Omega(g(n))$
- if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$, then $f(n) = \Omega(g(n))$ does not hold.

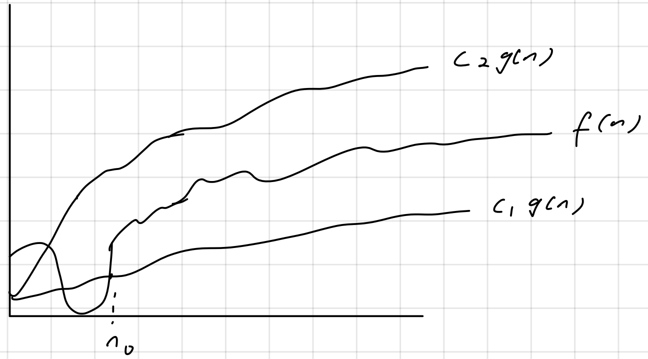
- Big-Θ notation

  Def: $f(n) = \Theta(g(n))$ if there are positive constant n_0, c_1 and c_2 such that for all $n \geq n_0$:

$$c_1 g(n) \leq f(n) \leq c \cdot g(n)$$

$f(n) \in \Theta(g(n))$

$f(n)$ grows like $g(n)$



Lemma [Big-Theta].

- If $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists, then $f(n) = O(g(n))$ $\Longleftrightarrow$

  $c_1 \leq \lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c_2$, where c_1 and c_2 are positive constant.

Corollary [Big-Theta].

- If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$, for some positive constant c, then $f(n) = \theta(g(n))$

- - - - - - - - - -

Another view:

Assume that $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists.

- $f(n) = O(g(n))$ if there exists $c > 0$ s.t.

  $$\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c$$

- $f(n) = \Omega(g(n))$ if there exists $c > 0$ s.t.

  $$\lim_{n \to \infty} \frac{f(n)}{g(n)} \geq c.$$

- $f(n) = \theta(g(n))$ if there exist $c_1, c_2 > 0$ s.t.

  $$c_1 \leq \lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c_2.$$

Useful Case:

If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$, then

$$f(n) \in O(g(n)) \quad \text{but} \quad f(n) \notin \theta(g(n))$$

If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$, then

$$f(n) \in \Omega(g(n)) \quad \text{but} \quad f(n) \notin \Theta(g(n))$$

If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c > 0$ $(c \neq \infty)$, then

$$f(n) \in \Theta(g(n)).$$

Note: Higher complexities are asymptotic upper bound for lower ones.

Some useful relations:

- For any two constants $a, b > 1$
  $\log_a n = \Theta(\log_b n) = \Theta(\lg n)$.
- $1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \Theta(n^2)$ (Arithmetic Sum).
- $1 + 2^2 + 3^2 + \cdots + n^2 = \sum_{i=1}^{n} i^2 = \Theta(n^3)$
- $1 + 2^d + 3^d + \cdots + n^d = \sum_{i=1}^{n} i^d = \Theta(n^{d+1})$
- $\lg 1 + \lg 2 + \cdots + \lg n = \lg n! = \Theta(n \lg n)$.
- $1 + \frac{1}{2} + (\frac{1}{2})^2 + \cdots + (\frac{1}{2})^m = \Theta(1)$ (Geometric Sum)
- For any $0 < r < 1$, $1 + r + r^2 + \cdots r^m = \frac{1 - r^{m+1}}{1 - r} = \Theta(1)$
- For any $r > 1$, $1 + r + r^2 + \cdots + r^m = \frac{r^{m+1} - 1}{r - 1} = \Theta(r^m)$.

Properties:

- $f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$
- Symmetry
  $f(n) = \Theta(g(n)) \Rightarrow g(n) = \Theta(f(n))$
  $f(n) = O(g(n)) \Rightarrow g(n) = O(f(n))$ Converse also holds.
- Transitivity:
  $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$ same for Ω and O.

(Assume all function are positive)

- $f(n) + g(n) = \theta(\max(f(n), g(n))$
- $f(n) + O(f(n)) = \theta(f(n))$
- if $f_1(n) = \theta(g_1(n))$ & $f_2(n) = \theta(g_2(n))$

  $\Rightarrow f_1(n) + f_2(n) = \theta(g_1(n) + g_2(n)) = \theta(\max(g_1(n) + g_2(n)))$
- if $f_1(n) = \theta(g_1(n))$ & $f_2(n) = \theta(g_2(n))$

  $\Rightarrow f_1(n) \times f_2(n) = \theta(g_1(n) \times g_2(n))$.

Best time complexit:

$T_{best}(n)$: best time of the algorithm over any input size n.

$T_{worst}(n)$: worst time of the algorithm over any input size n.

↑

Focus on worse-case complexity for analysis.

Expected Time Complexity

Expected /average running time:

$ET(n) = \sum_I P_r(I)\, time(I)$, for all case input I

$P_r(I)$ = probability of input I

$T(I)$ = running time of given input I

- An input probabilistic distribution model has to be assumed.
- For fixed input, running time is fixed.
- Average time complexity is for if we consider running it for a range of inputs, what the average behavior is.

Random Algorithm:

- No assumption in input distribution.
- Fixed input, running time is NOT fixed.
- Expected time is what we can expect when we run the algorithm on any single input.

From Probability:

$E(X) = \Sigma_I Pr(X=I) \cdot I$

linearity : $E(X_1 + X_2) = E(X_1) + E(X_2)$

Conbitional : $E(X) = E(X|Y)Pr(Y) + E(X|\text{Not } Y)(1 - Pr(Y))$.

ex).

```
k = ranbom(n)
for i in 1 to k
    for j in 1 to k
```

$ET(n) = \Sigma_{i=1}^{n} Pr(k=i)(ci^2) = \Sigma_{i=1}^{n} \frac{1}{n}(ci^2) = \frac{c}{n} \Sigma_{i=1}^{n} i^2 = \theta(n^2)$

ex)

```
k = ranbom(n)
if $k \leq \log n$ then
        for i = 1 to n:
```

Two cases, $k \leq \log n$ & $k > \log n$.

$Pr(k \leq \log n) = \frac{\log n}{n}$

$Pr(k > \log n) = 1 - \frac{\log n}{n}$

$ET(n) = Pr(k \leq \log n)\, T(k \leq \log n) + Pr(k > \log n)\, T(k > \log n)$

$= \frac{\log n}{n}(cn) + (1 - \frac{\log n}{n})(c')$

$= \theta(\log n)$.

Theoretical Lower Bounh f(n):

- if every possible algorithmis worst-case time complexiey is $\Omega(f(n))$.

A lower bounh $f(n)$ for problem-P is eight if there exist an algorithm for problem-P whose worst case running time is $\theta(f(n))$.

Search Problem

Binary search

In database, running multiple queries efficiently are neeｄeh.

Preprocessing time + Queries time

(time to prepare data for efficient query) (time to execute query).

ex) ① Brute force

preprocess: $O(1)$

search : $\theta(n)$

time : $O(1) + m \times \theta(n) = \theta(mn)$

if $m = n$: time $= \theta(n^2)$

② Pre-sort

preprocess: $\theta(n\log n)$

search : $\theta(\log n)$

time : $\theta(n\log n) + m \times \theta(\log n)$

$= \theta((n+m)\log m)$.

if $m=n$, time $= \theta(n\log n)$.

Binary search in soreeh array :

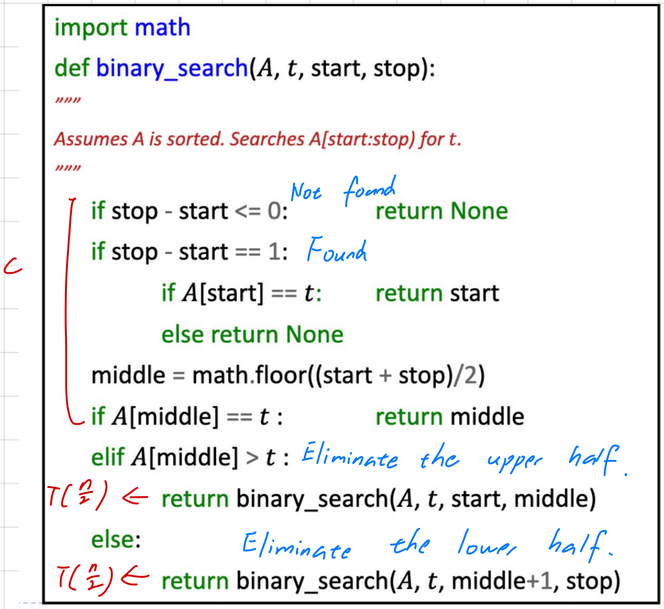
Input:

a sorteh array A whose elements are in non-decreasing order as indicies increase.

a target key t

Output:

return the index of A whose element equals to t.



```
import math
def binary_search(A, t, start, stop):
"""
Assumes A is sorted. Searches A[start:stop] for t.
"""
   if stop - start <= 0:   Not found   return None
   if stop - start == 1:  Found
         if A[start] == t:        return start
         else return None
c  middle = math.floor((start + stop)/2)
   if A[middle] == t :            return middle
   elif A[middle] > t : Eliminate the upper half.
T(n/2) ← return binary_search(A, t, start, middle)
   else:         Eliminate the lower half.
T(n/2) ← return binary_search(A, t, middle+1, stop)
```

$A = [0, 3, 5, 7, 9, 10]$

start = 0

stop = 5 $\quad t = 9$

① mid = 2

$A[mid] = 5$

$A[mid] < t$

$\begin{cases} start = mid + 1 = 3 \\ stop = 5 \end{cases}$

② mid = 4

$A[mid] = 9$

return 9.

Correctness of binary search.

① Base case:

stop − start ≤ 0 returns None

stop − start $= 1$, check A[start] and return.

② Recursive step, will it get terminated? (problems get smaller).

The algorithm will terminate as the problem get smaller till we reach base case.

③ Correctness:

Assume all recursive calls return correct answers.

by inductive argument, the algorithm return correct answer.

Best case complexity: $O(1)$

Worst case complexity:

Recurrence relation:
$$T(\overset{\text{stop-start}}{\overset{\downarrow}{n}}) = \begin{cases} T(\frac{n}{2}) + c & , n > 1 \\ \theta(1) & , n \le 1 \end{cases}$$

Solve recurrence relation for time complexity:

① Unroll several time

$$T(n) = T(\tfrac{n}{2}) + c$$
$$= (T(\tfrac{\frac{n}{2}}{2}) + c) + c \quad = T(\tfrac{n}{4}) + 2c$$
$$= T(\tfrac{n}{8}) + 3c$$
$$= \dots$$

③ Write general formula

$$T(n) = T(\tfrac{n}{2^k}) + kc \swarrow$$

determing the k in terms of input

or determing the # iteration in terms of input size.

③ Solve # of unrolls needed → solve k

stop when $\frac{n}{2^k} = 1$ unrolling terminates when reaching $T(1)$

$2^k = n$

$k = \log_2 n$.

④ Plug into general formula.

$$T(n) = T(\tfrac{n}{2^{\log_2 n}}) + \log_2 n \cdot c$$

$$= T(1) + c \log_2 n$$
$$= \theta(1) + \theta(\log_2 n)$$
$$= \theta(\log n).$$

Note: Theoretical Lower Bound (TLB) for searching in sorted list is $\Omega(\log n)$.

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

$$T(n) = T(\tfrac{n}{2}) + n$$
$$= T(\tfrac{n}{4}) + \tfrac{n}{2} + n$$
$$= \cdots$$
$$= T(\tfrac{n}{2^k}) + n + \tfrac{n}{2} + \cdots + \tfrac{n}{2^{k-1}}$$

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

So, $T(n) = T(1) + n(\underbrace{\tfrac{1}{2} + \cdots + \tfrac{1}{2^{\log_2 n - 1}}}_{O(1)})$

$$= \theta(n)$$

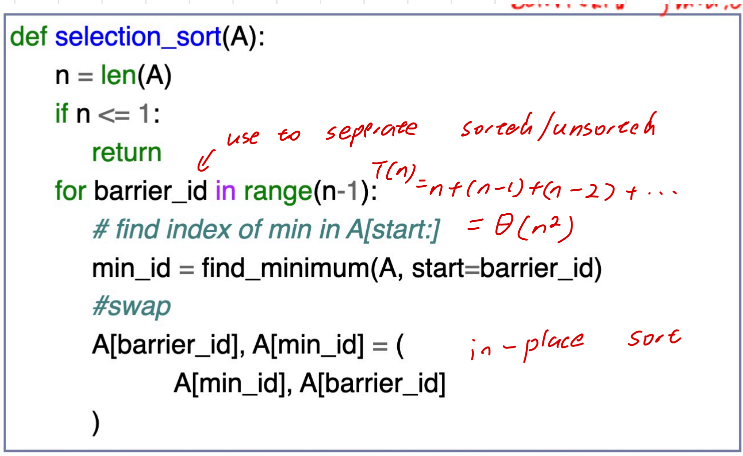
Sorting

Selection Sort

ideas: At each iteration, identify the smallest number in the remainder unsorted portion of array.

Put it at the end of the already-sorted portion.

Iterate till the end.



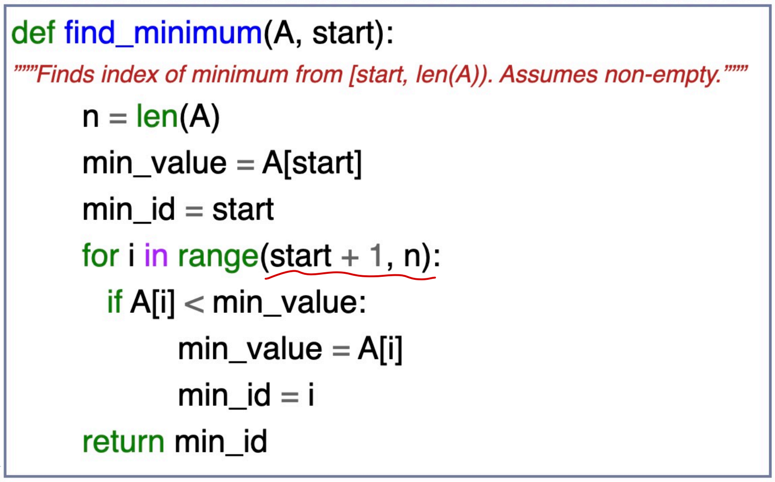
```
def selection_sort(A):
    n = len(A)
    if n <= 1:
        return
    for barrier_id in range(n-1):
        # find index of min in A[start:]
        min_id = find_minimum(A, start=barrier_id)
        #swap
        A[barrier_id], A[min_id] = (
                A[min_id], A[barrier_id]
        )
```

use to seperate sorted/unsorted

$T(n) = n + (n-1) + (n-2) + \cdots$

$= \theta(n^2)$

in-place sort



```
def find_minimum(A, start):
"""Finds index of minimum from [start, len(A)). Assumes non-empty."""
    n = len(A)
    min_value = A[start]
    min_id = start
    for i in range(start + 1, n):
      if A[i] < min_value:
          min_value = A[i]
          min_id = i
    return min_id
```

prove correctness using loop invariants

is a statement that holds at the end of each iteration, to show that it holds for each iteration, we first show it holds for the base case, then argue that if it holds at the end of (i-1)-th iteration, it will holds at the end of i-th iteration

(inductive ideas)

loop invariant: after k iterations,

the first k numbers in A are sorted, and are smaller than all the remainder n-k numbers.

if it holds for any k, then after k = n-1 iterations, we get a sorted array.

Base case: k = 0, loop invariant holds trivially

Inductive step: if it holds for k-1,

then we identify the smallest from the remainder n-k+1 numbers, which must be the k-th smallest of the original array.

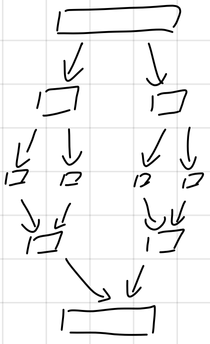
So after k-th iteration, loop invariant holds for k.

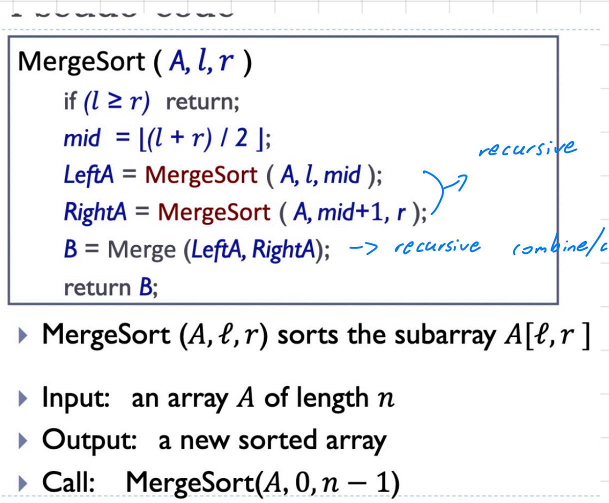
Time Complexity: $T(n) = cn + c(n-1) + c(n-2) + \cdots + c \cdot 1$

$= \Theta(n^2)$

Merge Sort

idea: divide-and-conquer, optimal worst case time complexity





Pseudo code

```
MergeSort ( A, l, r )
    if (l ≥ r)  return;
    mid  = ⌊(l + r) / 2⌋;
    LeftA = MergeSort ( A, l, mid );
    RightA = MergeSort ( A, mid+1, r );
    B = Merge (LeftA, RightA);
    return B;
```

recursive divide

-> recursive combine/conquer

- MergeSort (A, ℓ, r) sorts the subarray $A[\ell, r]$
- Input: an array A of length n
- Output: a new sorted array
- Call: MergeSort$(A, 0, n-1)$

Correctness:

① Base case: portion of array of size 1 is already sorted.

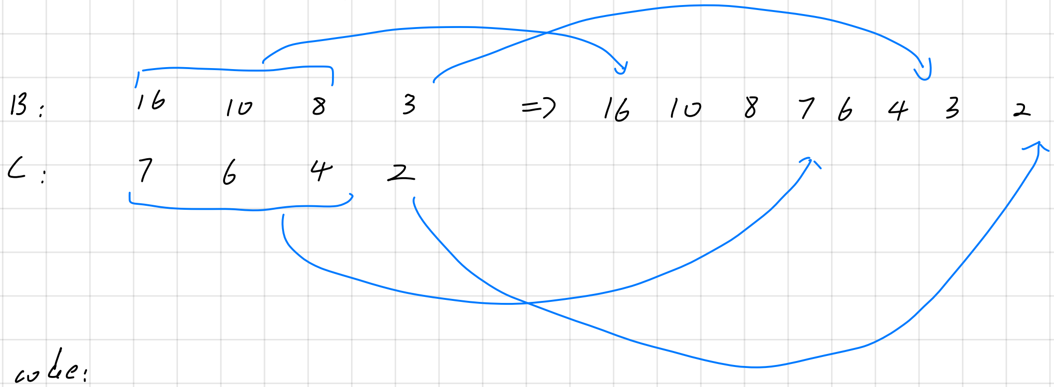
② Work on smaller problem and will terminated.

③ Recursive call return correct output, entire algorithm work.

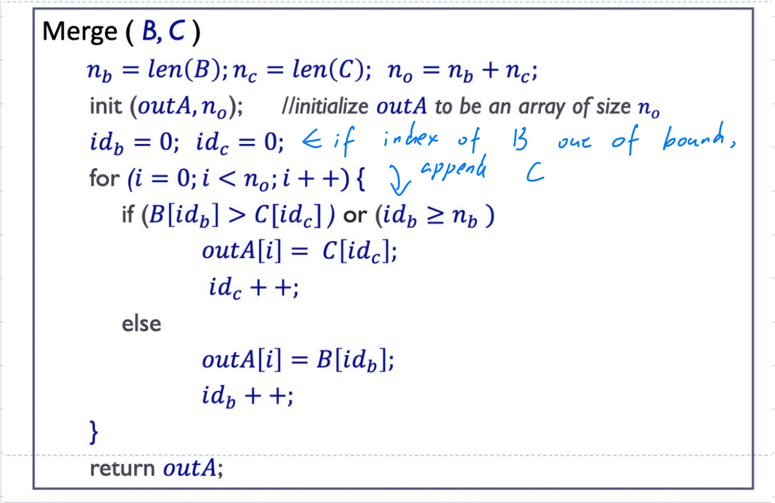
Conquer.

input: given two sorted array B and C

merge into a single sorted array



In code:



```
Merge ( B, C )
    $n_b = len(B); n_c = len(C);\ \ n_o = n_b + n_c;$
    init $(outA, n_o)$;      //initialize $outA$ to be an array of size $n_o$
    $id_b = 0;\ \ id_c = 0;$
    for $(i = 0; i < n_o; i++)$ {
        if $(B[id_b] > C[id_c])$ or $(id_b \geq n_b)$
                $outA[i] = C[id_c];$
                $id_c++;$
        else
                $outA[i] = B[id_b];$
                $id_b++;$
    }
    return $outA$;
```

← if index of B out of bound, (in case when all elemenes of B < c).

↓ append C

Time Complexity:

① Worst case time complexity for Merge(B, C).

$T_{merge} = \Theta(n_b + n_c)$, where n_b, n_c is length of B, C.

② Merge Sort:

from Merge

$$T(1) = T(\tfrac{n}{2}) + T(\tfrac{n}{2}) + cn = 2T(\tfrac{n}{2}) + cn$$

from recurrence relation of Merge Sort

$$T(n) = 2T(\tfrac{n}{2}) + cn$$

$$= 2(2T(\tfrac{n}{4}) + c\tfrac{n}{2}) + cn = 4T(\tfrac{n}{4}) + 2cn$$

$$= 4(2T(\tfrac{n}{8}) + c\tfrac{n}{4}) + 2cn = 8T(\tfrac{n}{4}) + 3cn$$

$$= \cdots$$

$$= 2^k T(\tfrac{n}{2^k}) + kcn.$$

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

$k = \log_2 n.$

so, $T(n) = 2^{\log_2 n} T(1) + c \cdot n \cdot \log_2 n$

$= n\,\theta(1) + cn\log_2 n$

$= \theta(n) + \theta(n\log n)$

$= \theta(n\log n)$

Note: merge sort not in-place.
has optimal asymptotic time complexity regardless of input shape.

Three-way Merge Sort

Divide into three arrays and then conquer.

Recurrence relation:

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

...

Quick Select

Order statistics: kth order statistics is the kth smallest number (or rank k).

ex) 1st order statistics : min

nth : max

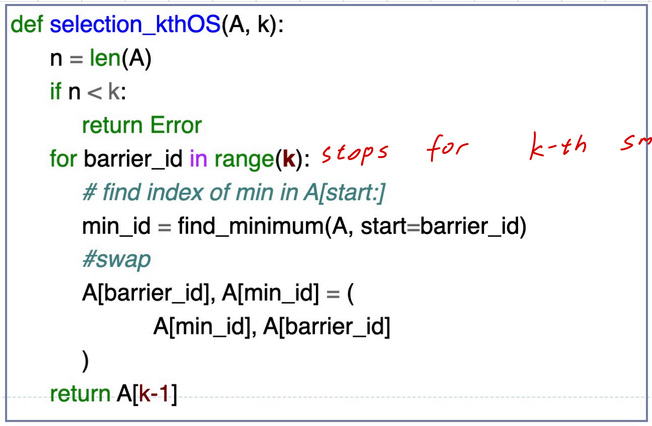
$\frac{n}{2}$-th : median

$\frac{pn}{100}$-th : p-th percentile, ...

Input: given n numbers in an array A.

Output: return the k-th order statistic of A.

Approach ①: modify selection sort



```
def selection_kthOS(A, k):
    n = len(A)
    if n < k:
        return Error
    for barrier_id in range(k):
        # find index of min in A[start:]
        min_id = find_minimum(A, start=barrier_id)
        #swap
        A[barrier_id], A[min_id] = (
                A[min_id], A[barrier_id]
        )
    return A[k-1]
```

stops for k-th smallest

$T(kn)$.

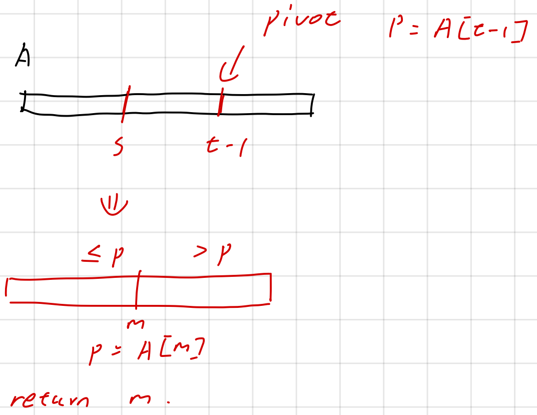
Approach ②:

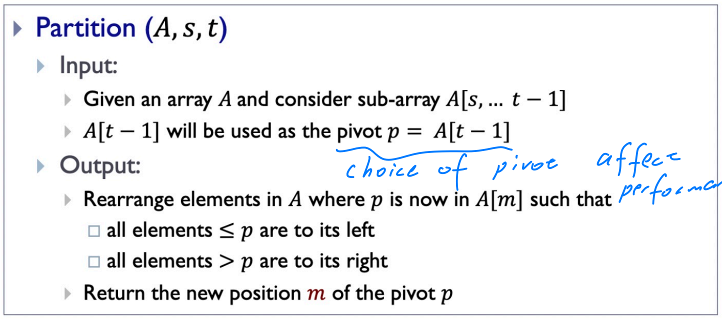
sort array A and return A[k].

$T(n\log n)$.

Approach ③: Quick Select:

1. Partition.





- **Partition (A, s, t)**
  - Input:
    - Given an array A and consider sub-array $A[s, \dots\ t-1]$
    - $A[t-1]$ will be used as the pivot $p = A[t-1]$
  - Output:
    - Rearrange elements in A where p is now in $A[m]$ such that
      - □ all elements $\le p$ are to its left
      - □ all elements $> p$ are to its right
    - Return the new position m of the pivot p

choice of pivot affect performance.

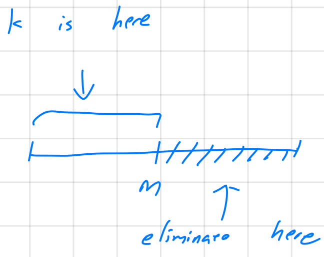
2. Quick select intution.

m = partition (A, 0, n)

case 1: k = m+1, return m

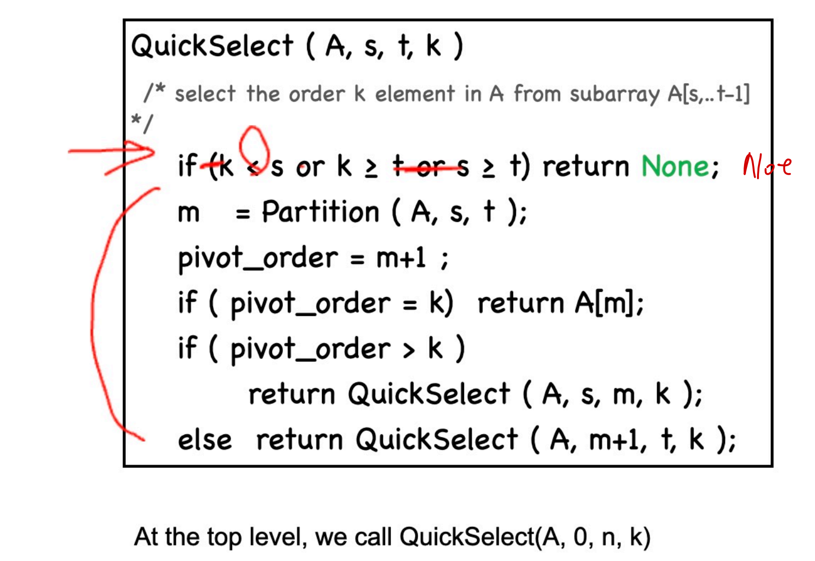
case 2: k < m+1, return QuickSelect (A, 0, m, k)

↑ m is too large, eliminate upper half.



Case 3: $k > m+1$, return QuickSelect(A, m+1, n, k)

↑ m is small, eliminate lower half.



```
QuickSelect ( A, s, t, k )
 /* select the order k element in A from subarray A[s,..t-1]
*/
    if (k < s or k ≥ t or s ≥ t) return None;   Not Found
    m   = Partition ( A, s, t );
    pivot_order = m+1 ;
    if ( pivot_order = k)  return A[m];
    if ( pivot_order > k )
          return QuickSelect ( A, s, m, k );
    else  return QuickSelect ( A, m+1, t, k );
```

At the top level, we call QuickSelect(A, 0, n, k)

In-place Partition(A, s, t).

idea: two moving indexes

One is to traverse the array and one is to swap smaller elements to the begining.

```
l = s
for r = s to t-2 do:
    if A[r] ≤ p then:
        swap A[l] with A[r].
        l++

swap A[l] with A[t-1]

return (l).
```

Complexity:

$\theta(t-s)$.

ex) $A = [13, 2, 5, 9, 4, 6]$

① Partition.

s ↓ l   t-1

1. [13, 2, 5, 9, 4, 6]

↑ r   ↑

l
↓

2. 13 2 5 9 4 6 A[r] ≤ p

↑
r

l
↓

2 13 5 9 4 6 swap

↑
r

3.

↓

2 13 5 9 4 6 A[r] ≤ p

↑

↓

2 5 13 9 4 6 swap

↑

↓

4. 2 5 13 9 4 6

↑

↓

5. 2 5 13 9 4 6 A[r] ≤ p

↑

↓

2 5 4 9 13 6

↑

6. 2 5 4 6 13, 9 swap A[l] and A[r-1]

← < 6 → ≥ 6.

done.

Time Complexity:

| r-1 | r | n-r |

↑

in each partition, we either enter the left part of array, or right part of array

$$T(n) = \underset{\substack{\uparrow \\ \text{why max?}}}{\max}(\overset{\text{left}}{T(r-1)}, \overset{\text{right}}{T(n-r)}) + cn. \quad , r = m+1 \text{ is pivot order.}$$

b/c we are considering the worst case

recursively, depends on value of r.

Lucky case -> eliminate exactly half each time.

$$T(n) = \max(T(\tfrac{n}{2}), T(\tfrac{n}{2})) + cn$$
$$= T(\tfrac{n}{2}) + cn$$
$$= \theta(n)$$

Worst case -> only eliminate one number at a time.

$$T(n) = T(n-1) + cn$$
$$= \cdots$$
$$= cn + c(n-1) + \cdots + c \cdot 1 = c(1 + \cdots + n) = c\,\theta(n^2) = \theta(n^2).$$

"Good split" is the one such that the subarray is in balanced.

e.g. pivot order $r \in [\frac{n}{4}, \frac{3n}{4}]$. ↘ this mean choose pivot whose rank is $[\frac{n}{4}, \frac{3n}{4}]$.

$$T(n) \le T(\tfrac{3n}{4}) + cn = \theta(n).$$

if always having good splits, then $T(n) = \theta(n)$.

↓

how to ensure good split

=>

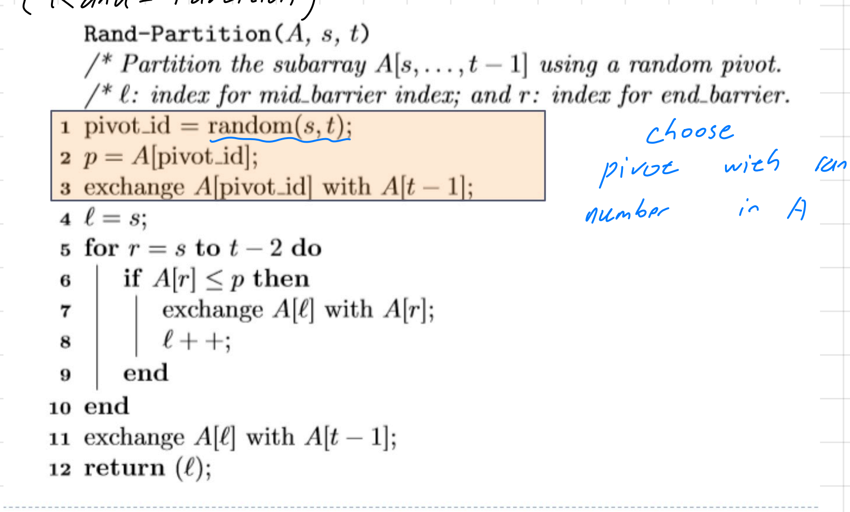
pick a random number $x \in A$.

why? b/c prob. of chooing any number in A is $\frac{1}{n}$.

$P(\text{if rank(chosen number)} \in [\frac{n}{4}, \frac{3n}{4}]) = (\frac{3n}{4} - \frac{n}{4})/n = 2/4 = \frac{1}{2}.$

=> means that in expectation, every two times is a good split.

(Rand-Partition)



```
Rand-Partition($A$, $s$, $t$)
/* Partition the subarray $A[s,\ldots,t-1]$ using a random pivot.
/* $\ell$: index for mid_barrier index; and $r$: index for end_barrier.
pivot_id = random($s,t$);
$p = A$[pivot_id];
exchange $A$[pivot_id] with $A[t-1]$;
$\ell = s$;
for $r = s$ to $t-2$ do
    if $A[r] \le p$ then
        exchange $A[\ell]$ with $A[r]$;
        $\ell{+}{+}$;
    end
end
exchange $A[\ell]$ with $A[t-1]$;
return ($\ell$);
```

choose pivot with random number in A

Expected Time Complexity intution.

- in expectation, after every constant number of calls, there will be "good split".
- "good split" will reduce problem size by at least $\frac{1}{4}$.

$$T(n) = \max(T(r-1), T(n-r)) + cn.$$

$$T_{good}(n) \le T_{good}\left(\frac{3n}{4}\right) + cn$$

$$= cn + \frac{3}{4}cn + \left(\frac{3}{4}\right)^2 cn \cdots$$

$$= cn\left(\underbrace{1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \cdots}_{\text{some constant}}\right)$$

$$= \Theta(n).$$

$P(\text{good split}) = \frac{1}{2}$

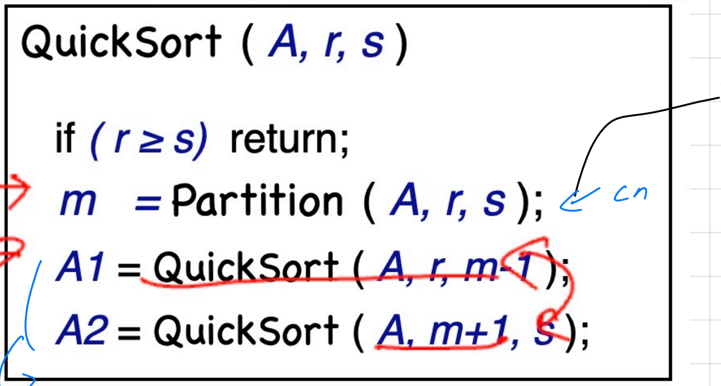
Expected cost of bad split bounded by $\left(\frac{1-p}{p}\right) T_{good}(n) = T_{good}(n).$

Exepected total complexity $ET(n) \le 2T_{good}(n) = \Theta(n).$

Randomized Quick Sort:

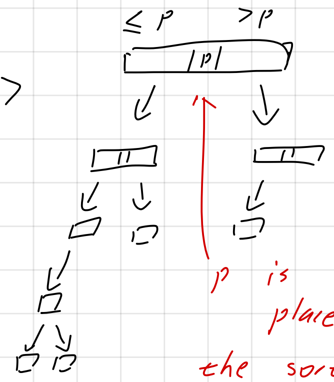
- In-place sort
- expected time: $\Theta(n\log n)$

worst case: $\Theta(n^2)$.



```
QuickSort ( A, r, s )
    if ( $r \ge s$ ) return;
    m = Partition ( A, r, s );        cn
    A1 = QuickSort ( A, r, m-1 );
    A2 = QuickSort ( A, m+1, s );
```

$T(n) = T(m-1) + T(n-m) + cn.$



much like merge sort but not a balanced tree.

p is in the correct place in relation to the sorted array, so keep it unmoved.

Worst case: $T(n) = T(n-1) + cn = \Theta(n^2)$

Best case: $T(n) = 2T\left(\frac{n}{2}\right) + cn = \Theta(n\log n)$

Expected: $ET(n) = \Theta(n\log n).$

Compared to Merge Sort

- inplace sorting
- faster practically (b/c a tree with less node, a almost sorted array require less swap).

Binary Search Tree

BST

Each node has at most 2 children.

Each node contains at least (Key, Left, Right, Parent)

A node is root if no parent.
A node is leaf if no children.

Complete binary tree

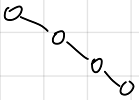
- Each node has two children
- Each level is filled, all nodes are as left as possible.

BST property:

$x.key \geq y.key$ if y is in left subtree of x.

- - - $\leq$ - - - if - - - right - - -

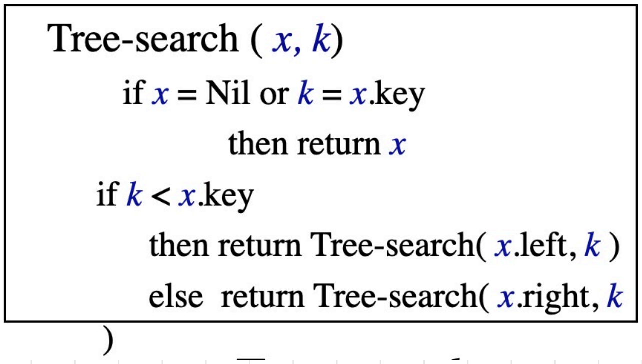
Tallest BST = n



Shortest BST = $\log_2 n$

Operation in BST:

① Tree-Search (root, k) - search for key k in tree x.



```
Tree-search ( x, k)
    if x = Nil or k = x.key
        then return x
  if k < x.key
      then return Tree-search( x.left, k )
      else  return Tree-search( x.right, k
)
```

Complexity : $T(n) = \Theta(\text{tree height}) = O(n) = \Omega(\log n)$.

② Find minimum / maximum.

```
Tree_min (x)
  while (x.left ≠ None):
      do   x = x.left.
return x.
```

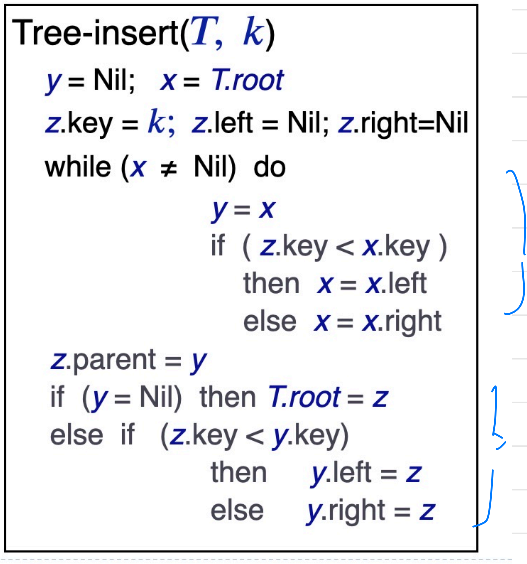
→ get the left most child node

⇓

Complexity: $T(n) = \Theta(h)$, where h is height of tree.

③ Tree_insert (x, k)

insert k to the tree such that resulting tree is still BST.



```
Tree-insert(T, k)
   y = Nil;   x = T.root
   z.key = k;  z.left = Nil; z.right=Nil
   while (x ≠ Nil)  do
             y = x
             if  ( z.key < x.key )
                then  x = x.left
                else  x = x.right
   z.parent = y
   if  (y = Nil)  then  T.root = z
   else  if   (z.key < y.key)
             then      y.left = z
             else      y.right = z
```

tree search, locate potential parent y for z.

$\Theta(h)$

set z as child of y.

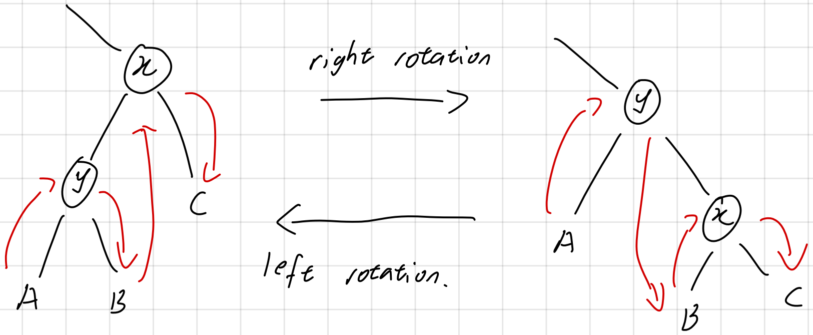
$\Theta(1)$

Balanced BST

Good tree such that height $h = \log n$.

by balancing the tree while doing insertion and deletion

Rotation technique to keep tree height low.



order is maintained.

In balanced BST, operation can be done in $\Theta(\log n)$

Select queries

BST-Select:

given a list of records whose keys are stored in a tree rooted at x.
return the node whose key has rank k.

Why not QuickSelect?

=> may do it many times and need a data structure that support Select under dynamic change.

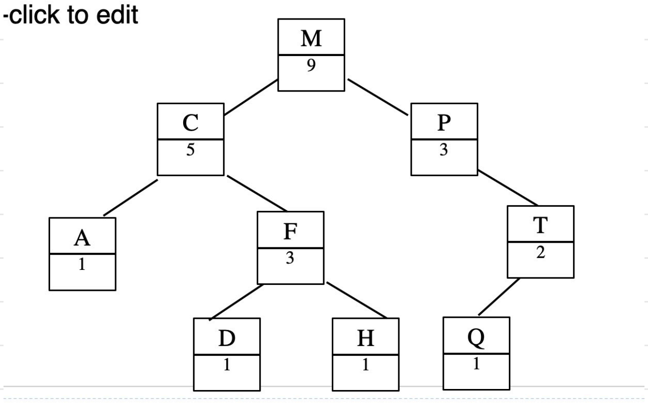
How?

By argument the rank in a BST.

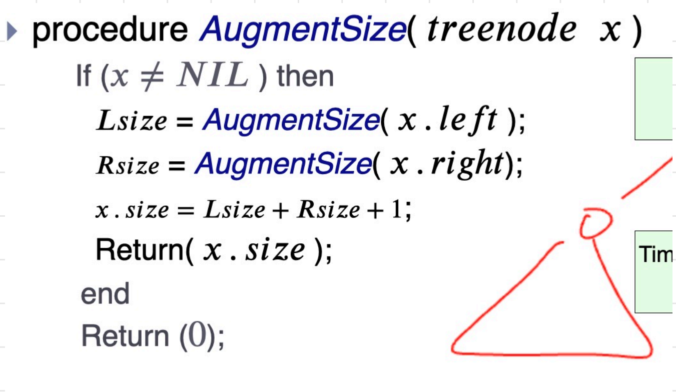
store $x.size$ = # nodes in the subtree.
space needed is $\Theta(n)$

$x.size = x.left.size + x.right.size + 1$

-click to edit



↓ Algorithm



```
procedure AugmentSize( treenode x )
    If ($x \neq NIL$) then
        $Lsize$ = AugmentSize( $x.left$ );
        $Rsize$ = AugmentSize( $x.right$ );
        $x.size = Lsize + Rsize + 1$;
        Return( $x.size$ );
    end
    Return (0);
```

Thus, we can implement BST-Select in $\theta(\log n)$ time, faster than $\theta(n)$. and support dynamic operation.

Hashing

Hash function: $f: \mathcal{U} \to X$ from one set to another.

- Deterministic
- "uniform" mapping and few "collisions".

Hash Table

Given a universe of elements $\mathcal{U}$.

Need to store some keys and perform insert / search / delete.

Membership queries and dynamic updates

Approach 0: use array to organize all keys.
pre-sort the array

Approach 1: organize keys in doubly-linked list.

Approach 2: organize keys in balanced BST

Approach 3: Direct address table (DAT).

Initialize table length to be 0 to all keys.

ex). keys are from 0 to 99999 for zip code.

Not memory efficient

⇓

Hash table :

- U : universe
- $T[0 \cdots m-1]$: a hash table of size m.
  - $m << |U|$
  - m to be around size of data.
- Hash function.

Mapping : $h : U \to \{0, 1, \dots, m-1\}$.

h maps each element in universe to an index in the hash table.

- $h(k)$ is the hash value of key k.

↳ store k in location $h(k)$ of hash table T.

- Collision :

Multiple keys hash to the same slot

Collision happens when $\underline{h(x) = h(y) \text{ for } x \neq y \in U}$.

Handle collision :

- Chaining

chain a linked list of stored elements that hash to j.

- open address

Operation :

- Chained-hash-insert

$O(1)$, insert x at the head of list $T[h(x)]$.

- Chained-hash-search

$O(len(T[h(x)]))$.

- Chained-hash-delete

  $o(\text{len}(T[h(x)]))$.

Good Hash function spread elements into table uniformly.

average case:

n # elements

m size of table

Load Factor $\alpha = \frac{n}{m}$ (average # of elements per linked list)

$\theta(n)$ worst case complexity.

Simple uniform hashing assumption:

any given elements is equally likely to hash into any of the m slots in T.

Let n_j be length of list $T[j]$.

$n = n_0 + n_1 + \dots + n_{m-1}$

under assumption

$E[n_j] = \alpha = \frac{n}{m}$

How?

- $\{k_1 \dots k_n\}$ set of keys
- $X_i = 1$ if $h(k_i) = j$

  0 otherwise
- $n_j = \Sigma_{i=1}^{n} X_i$
- $E[X_i] = P[h(k_i) = j] = \frac{1}{m}$.
- $E[n_j] = E[\Sigma_{i=1}^{n} X_i] = \Sigma_{i=1}^{N} E[X_i] = \frac{n}{m}$.

Under assumption,
expected running time:

- Search

  $ET(n) = \theta(1 + \frac{n}{m})$

  worst case $T(n) = \theta(n)$

- Insert

  $T(n) = \theta(1)$

- Delete

  $ET(n) = \theta(1 + \frac{n}{m})$

  worst case $T(n) = \theta(n)$.

if $\alpha = \frac{n}{m} = \theta(1)$,

the operation take $\theta(1)$ time.

Downside of Hashing

- Only support dictionary queries
  membership query + insert / delete
  - cannot query multiple elements whose total is close to something
  - canot do range query
- No locality

Graphs

Graph $G = (V, E)$.

V: a set of graph node (or vertices).

$E \subset V \times V$: a set of graph edges.

- each edge $(a, b) \in E$ represents a certain relation between the pair of graph nodes, $a, b \in V$.

- Directed Graphs:

  V is a finite state of nodes

  E is a set of ordered pairs called edges.

  - $(a, b) \neq (b, a)$.
  - may be self loop (a, a).
  - simple graph: for any ordered pair, there can be at most one edge in E.

- Undirected Graph

  V is a finite set of nodes

  E is a set of unordered pairs

  - $\{a, b\}$, edge is subset of nodes V with cardindity 2.
  - No order for each pair.

$(a, b) = (b, a)$.

- Simple Graph:
  - No self-loops
  - At most one edge for each pair of nodes.

	Edge direction	Self loop	Opposite edges (a,b) & (b,a).
Directed	Yes	Yes	Yes
Undirected	No	No	No.

- Given an undirected graph $G = (V, E)$.
  - given edge $e = (u, v) \in E$, u, v is end-point of e.
  - edge e is incident on node u if u is an end-point of e.

- Given undirected graph $G = (V, E)$, the degree of a node $v \in V$ is
  - $\deg(v) :=$ number of edges incident on v.

- Given undirected graph $G = (V, E)$ with $n = |V|$.
  - $0 \leq \deg(v) \leq n-1$
  - $\Sigma_{v \in V} \deg(v) = 2|E|$
  - maximum numbe of edge is $\dfrac{n(n-1)}{2}$
  - $|E| = O(n^2)$.

- Undirected graph is complete graph $\Leftrightarrow$ there is one edge between every pair of distinct nodes in V.

  $|E| = \dfrac{n(n-1)}{2}$ (fully connected)

- Given a directed graph $G = (V, E)$.

in-degree $(v) :=$ # of edges entering v

out-degree $(v) :=$ # of edges leaving v

$degree(v) = indeg(v) + outdeg(v)$

- Given a directed graph $G = (V, E)$, with $n = |V|$

  $0 \leq indeg(v), outdeg(v) \leq n$, for any node $v \in V$

  $\Sigma_{v \in V} indeg(v) = \Sigma_{v \in V} outdeg(v) = |E|$.

  $|E| = O(n^2)$

- Given undirected graph $G = (V, E)$.

  the set of neighbors of $v \in V$ is the set of all nodes in V that share an edge with v.

- Give directed graph.

  the set of successors is the set of nodes at the end of an edge leaving v

  the set of predecessors is the set of nodes at the start of an edge entering v,

- Path: from u to u' is a sequence of one or more nodes $u = v_0 \cdots v_k = u'$ s.t there is an edge between each consecutive pair of nodes in sequence.

  Length of path = # of nodes -1 = # of edges in a path.

- Path is simple if it visit each node once.
- A cycle is a path where the first and last nodes are same
- Node u is reachable from node v if there is a path from v to u.
  - in undirected graph, reachability is symmetry, u is reachabe from v $\Leftrightarrow$ v is reachable from u.
  - in directed graph, reachability is not symmetry.

- Connectivity, for undirected graph is connected if every node is reachable from every other node. Otherwise, it is disconnected
- Connected component is a maximally-connected subset of nodes of V.
  - given undirected graph, it is a set $C \subseteq V$ s.t.

    1. any pairs $u, v \in C$ are reachable from one another &
    2. if $u \in C$ and $z \notin C$, the u and c not reachable.

  - connected $\Rightarrow$ only 1 connected component.

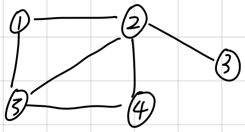
Graph Representation:

① Adjacency matrix

assume $V = \{v_0, v_1, \dots, v_{n-1}\}$, $n = |V|$.

adjacency matrix of a graph is a $n \times n$ matrix

$$adj[i,j] = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$

if undirected graph, symmetry adj.

size : $\Theta(|V|^2)$

edge query : $adj[i,j] == 1$ $\Theta(1)$

degree (i) : np.sum$(adj[i,:])$ $\Theta(|V|)$

Pro :

- support efficient edge queries
- easy to use
- easy to manipulate
  - (i,j)-th entry of A^2 gives number of hops of length 2 between v_i & v_j

Con :

- take too much space $\Theta(|V|^2)$. especially for sparse graph.

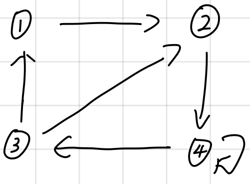
② Adjacency List

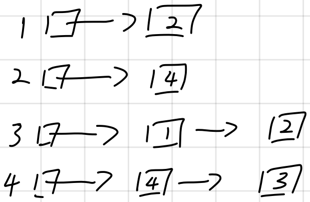
Each vertex u has a list, recording its neighbor.

⇒ An array of $|V|$ lists.

$adj[i]$.size = size of AdjList for node v_i

$adj[i]$ = adjacency list for node v_i





Size : for each vertex v_i, adjacency list $adj[i]$ has size =

deg(v_i) if undirected (each edge stored twice)

outdeg(v_i) if directed. (each edge stored once) so size = $\Theta(|V|+|E|)$,

where $\Theta(|V|)$ for outer array
$\Theta(|E|)$ for total length of edges.

edge query : j in adj[i] $\Theta(\text{degree}(i))$

degree : len(adj[i]) $\Theta(1)$

Pro :

- optimal space
- Fast for degree query

Con :

- slow for edge query
- No linear algebra manipulation.

③ Dictionary - set

change the inner list to set and outer query to hash table. (dictionary)

size : $\Theta(|V|+|E|)$.

edge query : j in adj[i] $\Theta(1)$

degree : len(adj[i]) $\Theta(1)$.

BFS

Graph search:

Each node has three states:

- undiscovered.
- pending (discovered but not explored).
- visited (done exploring).

At beginning, all nodes are undiscovered.

- Search will choose next node to visit (explore) from list of pending node.
- If node is "visited", then all neighbors should be "pending" or "visited".

BFS: choose the "oldest" pending nodes

BFS(G, s).

idea :
- all nodes are undiscovered, other than source node, initialized as pending
- At each step:
  - take the oldest pending node to explore
  - mark all its undiscovered neighbors as pending.
  - mark this node as "visited"
- Repeat until no more pending nodes

Implementation: FIFO data structure (queue) for pending list.

- Enqueue(a)
- Dequeue(a)

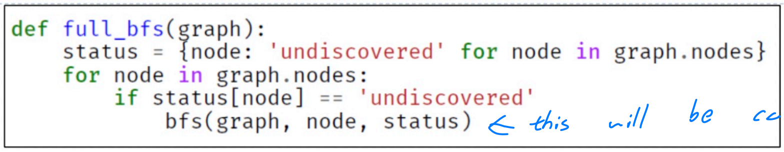
} → $\Theta(1)$ complexity.

Use array/hash table to store status.

BFS will visit the set of nodes reachable from source node.

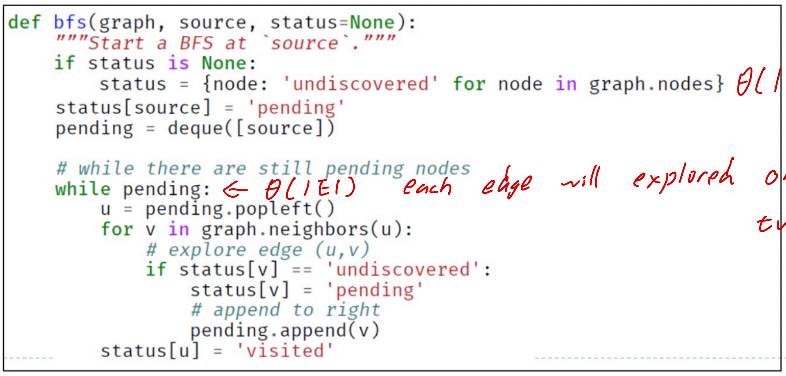
↓

Full BFS (visit all nodes).



```
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered':
            bfs(graph, node, status)
```

← this will be called k times for k connected components.



```
def bfs(graph, source, status=None):
    """Start a BFS at `source`."""
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}
    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

$\Theta(|V|)$.

← $\Theta(|E|)$ each edge will explored once for directed graph

twice for undirected graph.

Complexity : $\Theta(|V|+|E|)$.

For BFS, complexity is $\Theta(|V|+m_s)$, where $m_s = \#$ edges in component of G containg source s.

(On a connected component)

and $m_s = O(|E|)$, upper bounded by # of all edges.

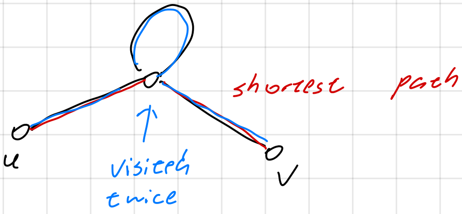
Shortest Path for BFS

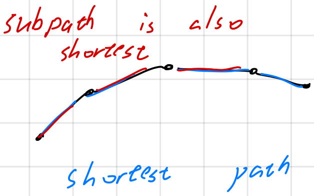
length of path is (# nodes in path -1).

Shortest path from u to v is a path from u to v with smallest possible length

Shortest path distance is length of shortest path.

Property : • Given any $u, v \in V$, if v is reachable from u, shortest path from u to v has to be simple.





- Any subpath of shortest path must be a shortest path.
- a shortest path of length k consists of a shortest path of length $(k-1)$ + 1 edge.

Find shortest path from BFS:

- Stat from source.
- Find all nodes that are distance 1 from s.
- Use them to find nodes distance 2 from s,

...

- Till we find all reachable nodes.

Intuitively:

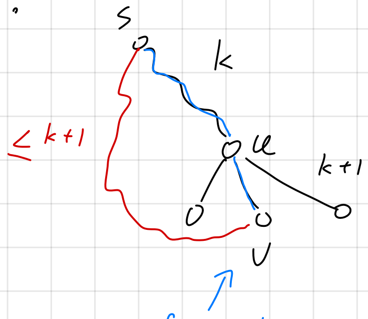
- The first time we discover a node encodes the fastest way to reach it.

Property of BFS:

For any $k > 0$,

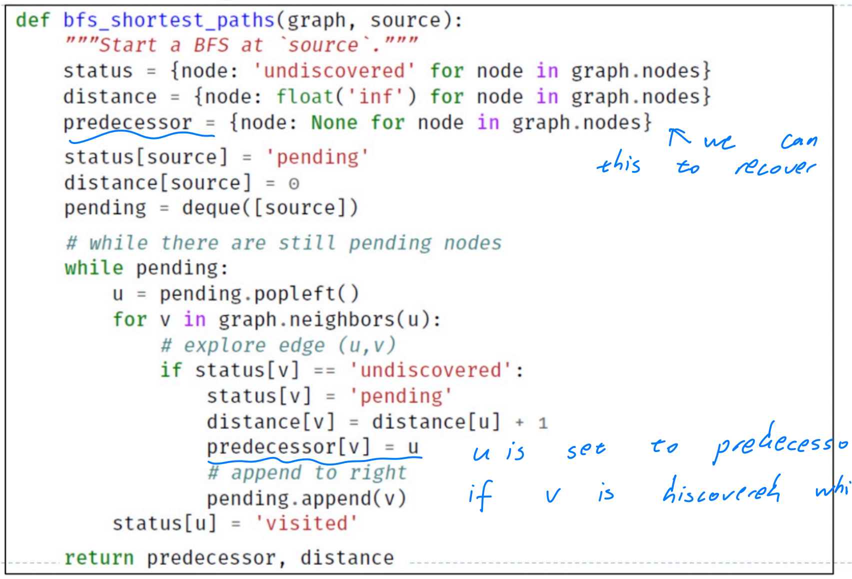
- all nodes at distance k from sources are added to the "pending" queue before any node of distance $> k$.
- nodes are "processed" in order of distance from the source.

↳ guarantee that the first time find a undiscovered node must be the shortest path to reach the node.



if v is undiscovered, then this path is shortest, with distance $k+1$

if v is already discovered, then there exist a shortest path s.t distance must $\leq k+1$.



```
def bfs_shortest_paths(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    distance = {node: float('inf') for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}
    status[source] = 'pending'
    distance[source] = 0
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                distance[v] = distance[u] + 1
                predecessor[v] = u
                # append to right
                pending.append(v)
        status[u] = 'visited'
    return predecessor, distance
```

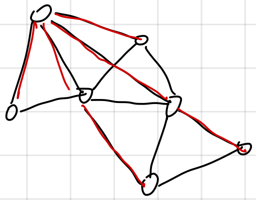
↖ we can use this to recover shortest path

u is set to predecessor of v if v is discovered while visiting u.

Same complexity as BFS

$\Theta(|V|+|E|)$.

Recover shortest path from BFS → BFS tree



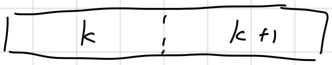
- Tree is connected graph $T = (V, E)$, $|E| = |V| - 1$
- Any two nodes in a tree, there is only one shortest path connecting them

=> Given a BFS-tree from source s, for the unique path from s to u in T is a shortest path in G, its length is shortest path distance.

Full BFS will give us a collection of BFS-tree, called forest

At any moment of BFS:

- the shortest path distance from source in queue are non-decreasing
- the shortest path distance for nodes in queue not diff more than 1.



the queue.

DFS

choose the "newest" pending nodes

idea:

all nodes initialized as undiscovered.

At each step:

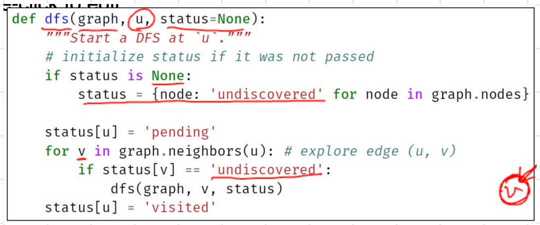
take the newest pending node
explore all undiscoverable nodes reachable
then mark this node as visited

Repeat untill no pending nodes.

Data Structure: Stack FILO.

↓ Implemented as

Recursive Algorithm

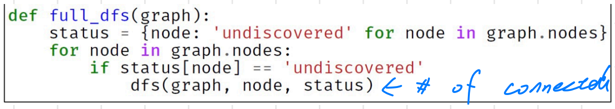


```
def dfs(graph, u, status=None):
    """Start a DFS at `u`."""
    # initialize status if it was not passed
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

DFS will visit all nodes reachable

Complexity: $\Theta(|V|+|E|)$. for full DFS.



```
def full_dfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            dfs(graph, node, status)
```

← # of connected component times execute

For each node v, DFS-predecessor is node u where through exploring edge (u,v) Node v was first discovered. (status to pending).

↓

Collection of edges of the form (predecessor(v) , v) give DFS-tree

Start & Finish time:

Node status change from undiscovered to pending: ⟶ Start time

=> first time this node is discovered

from pending to visited: ⟶ Finish time.

=> exploration of this node is finish.

(all neighbors are visited except predecessor)

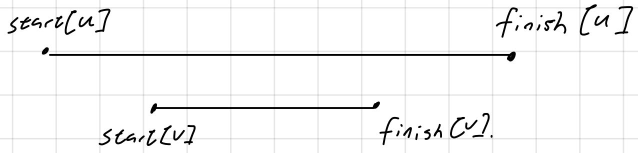
(Increment by 1 when some nod marked as pending / visiting).

Property:

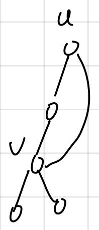
① Take any two nodes u and v. Assume Start[u] ≤ Start[v].

Exactly one of the following two is true:

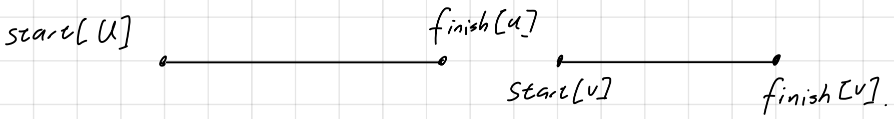
- start[u] ≤ start[v] ≤ finish[v] ≤ finish[u]

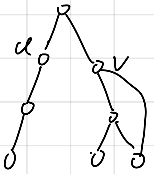


explore all reachable from u before finish u.



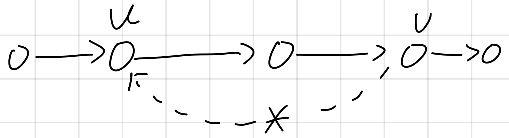
- $start[u] \le finish[u] \le start[v] \le finish[v]$.





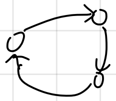
② If node v is reachable from u, but u is not reachable from v

then $finish[v] \le finish[u]$.

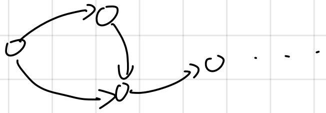


Topological Sort:

Directed cycle is a (directed) path from a node to itself.



Directed acyclic graph (DAG) is a directed graph that does not contain any directed cycle.

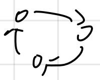


Given a DAG, $G = (V, E)$, topological sort of G is an ordering of V s.t. for an edge $(u, v) \in E$, u comes before v in ordering.

Topological sorts of same DAG are not unique.

Claim: directed graph $G = (V, E)$ a topological sort $\Leftrightarrow$ G is DAG.

b/c if there is a cycle, no valid ordering for nodes



(who comes first?)

(nodes with later finish time should come first).

Topo-sort Algorithm:

- First perform DFS $\rightarrow \Theta(V+E)$
- Output the order in decreasing order of finish time. $\rightarrow \Theta(V)$.

Bellman-Ford

Weighted graph $G=(V,E;w)$.

is a graph $G=(V,E)$ with edge weight map $w: E \rightarrow \mathbb{R}$.

(can be directed or undirected.)

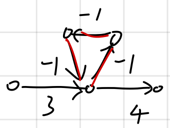
Path length: total weight of all edges in path.

A shortest path from u to v is a path from u to v with minimum length.

A shortest path may not be unique, but all with same length.

Shortest path is not well defined if there is "negative cycles".

↓ a cycle whose length is negative.



↓

Assume no "negative cycle",

then there is always a shortest path that is simple (no cycle at all)

Theorem:

Optimal Substructure Property:

If $(u_1, u_2, \ldots, u_m)$ is a shortest path from u_1 to u_m, then any sub-path $(u_i, \ldots, u_j)$ is also a shortest path.

Let $\delta(u,v)$ denote shortest path distance from u to v.

Suppose (z,v) is an edge, then:

$$\delta(s,v) \le \underbrace{\delta(s,z)+w(z,v)}_{\text{shortest path from } s \text{ to } v \text{ using edge } (z,v) \text{ as last edge.}}.$$

And if $\delta(s,z) = \delta(s,z)+w(z,v)$, then z is predecessor of v along with shortest path s to v.

Single-source shortest path (SSSP) problem:

Given weighted graph $G=(V,E;w)$ and source node s, compute the shortest path distance from s to all other nodes in V.

BFS work for unweighted graph, but not weighted graph with difference edge weight.

Edge Update

Bellman-ford work for any weighted graph

complexity $\Theta(V \cdot E)$

Dijkstra work for graph with positive edge weight.

complexity $\Theta((V+E)\lg V)$ and can be make to be $\Theta(V\lg V+E)$.

Both use update() operation

idea: both algorithm keep track of the shortest path found so far (estimated shortest path).

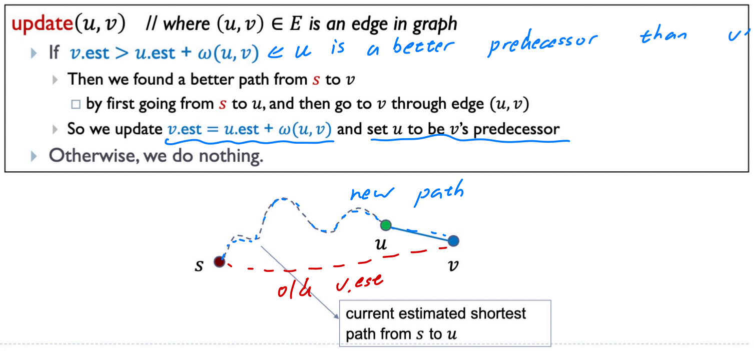
set $u.est$ = length of estimated shortest path source s to u.

At beginning $u.est = \infty$ & $s\_est = 0$, iteratively update estimate.

Key: • during update process

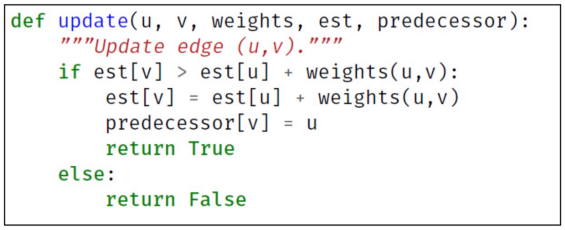
- estimated shortest path can only improve
- at least as long as true shortest path
- once found shortest path, it will not changed

- For each node u, we keep u's
  - predecessor along the shortest path from s to u
  - u.est, current estimated distance.



update(u, v) // where $(u, v) \in E$ is an edge in graph

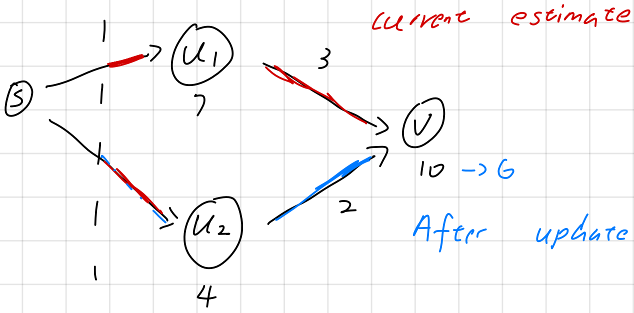
- If v.est $> u$.est $+ \omega(u, v)$ ⇐ u is a better predecessor than v's current predecessor
  - Then we found a better path from s to v
    - □ by first going from s to u, and then go to v through edge (u, v)
  - So we update v.est $= u$.est $+ \omega(u, v)$ and set u to be v's predecessor
- Otherwise, we do nothing.



```
def update(u, v, weights, est, predecessor):
    """Update edge (u,v)."""
    if est[v] > est[u] + weights(u,v):
        est[v] = est[u] + weights(u,v)
        predecessor[v] = u
        return True
    else:
        return False
```

$\theta(1)$

update (u_2, v) :



Theorem:

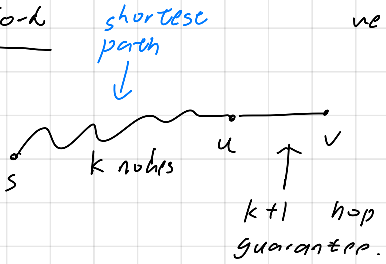
Let u & v be node.

Suppose
- current shortest path u.est is correct.
- there is shortest path from s to v, with u being v's predecessor.

⇒ After update(u,v), estimated shortest path distance to v is correct.

Bellman-Ford

we can compute k+1 hop via update() if k hops shortest path are found.



Note: if v.est is correct, then any further update will not change v.est.

Algorithm: perform update for all edges in E iteratively.

Loop invariant: • suppose we perform "update all edges" k times.

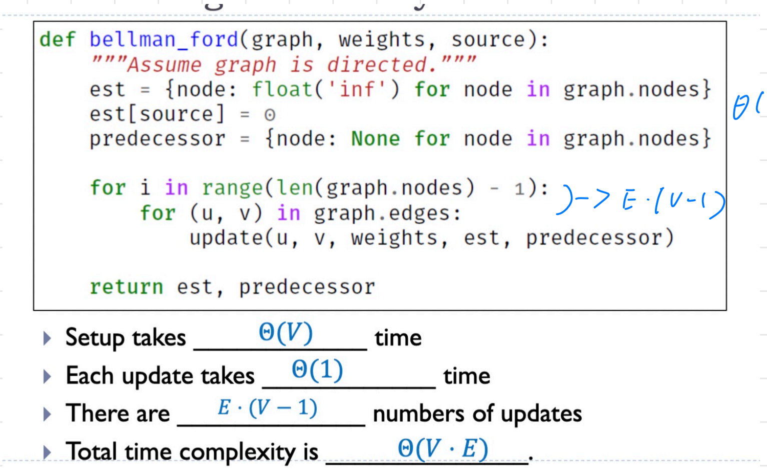
||

⇓

All nodes whose shortest path from source s has ≤ k edges are guaranteed to estimate correctly.

⇓

perform V-1 times to guarantee correct for any graphs.



```
def bellman_ford(graph, weights, source):
    """Assume graph is directed."""
    est = {node: float('inf') for node in graph.nodes}   Θ(V)
    est[source] = 0
    predecessor = {node: None for node in graph.nodes}

    for i in range(len(graph.nodes) - 1):   )-> E·(V-1)
        for (u, v) in graph.edges:
            update(u, v, weights, est, predecessor)

    return est, predecessor
```

- Setup takes $\Theta(V)$ time
- Each update takes $\Theta(1)$ time
- There are $E \cdot (V-1)$ numbers of updates
- Total time complexity is $\Theta(V \cdot E)$.

Early stopping: no distance change for all edges in a round ⇒ early stopping

Decect negative cycles: after V iteration of Bellman-ford, if estimated distance still decreasing, mean there is a negative cycle.

Dijkstra Algorithm

Not all path need to updated in each round.

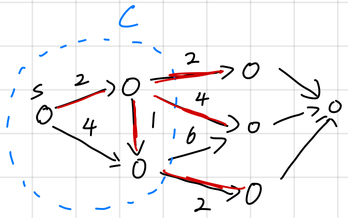
idea: the algorithm explore the nodes in a greedy manner, in increasing distance to the source.

↓

by the time we explore a node, algorithm guarantee to have correse estimated distance.

- Keep track of a set of C of correct nodes.
- At every step, add node outside of C with smallest estimated distance to C; update estimated distance to its neighbor.

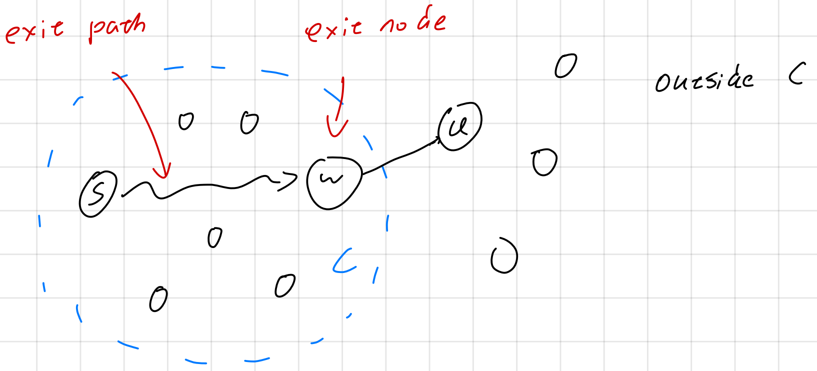
ex)



Exit Path:

An exit path through C is a path $\pi: s \rightsquigarrow u$ from the source s to some node $\mu \notin C$, called exit node, such that π consist of:

- a path in C from s to some node w.
- followed by an edge (w, u) (exit edge) to reach exit node u.



(an exit path from s) + (path from exit node to u)

Loop invariant:

- At beginning of each while loop, distance in C is correct.
- For each node u outside C, u.est store the length of shortest exit path to u.

proof:

- consider path π from s to u. Let y be exit node.

$(s \text{ to } u) \geq (s \text{ to } y) + (y \text{ to } u)$.

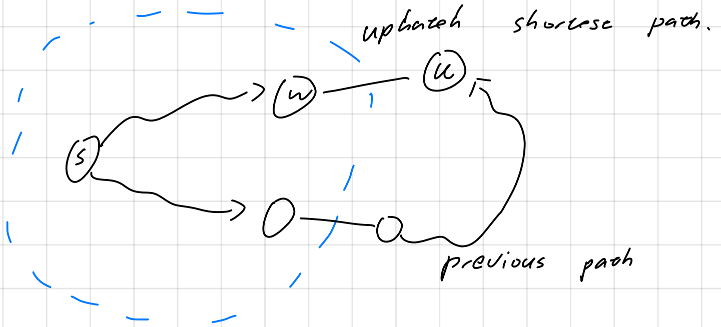
- since $(y \text{ to } u) \geq 0$.

$\Rightarrow (s \text{ to } u) \geq (s \text{ to } y) + 0$

$\geq$ length of shortest path from s to y

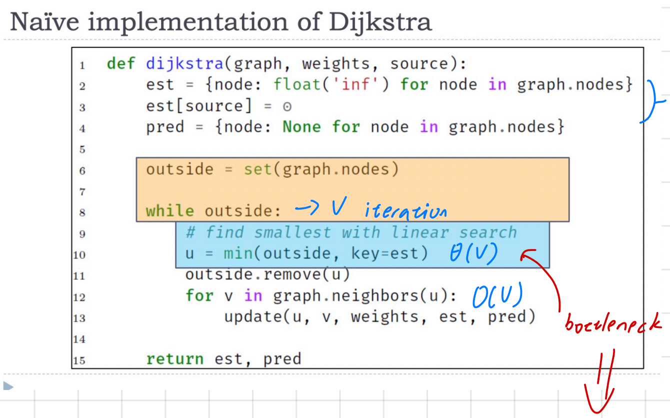
$= y.est \geq u.est$

$\Rightarrow (s \text{ to } u) \geq u.est \Rightarrow$ u.est must be shortest path distance.



· After while loop $C' = C \cup \{u\}$, then update neighbor of u.

using see



Naïve implementation of Dijkstra

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    outside = set(graph.nodes)

    while outside:  → V iteration
        # find smallest with linear search
        u = min(outside, key=est)  θ(V)
        outside.remove(u)
        for v in graph.neighbors(u):  θ(V)
            update(u, v, weights, est, pred)

    return est, pred
```

$\} \to \theta(V)$

complexity:

$\theta(V) + \theta(V) \times V = \theta(V^2)$.

bottleneck

⇓

Solution: Priority Queue.

extract and delete min

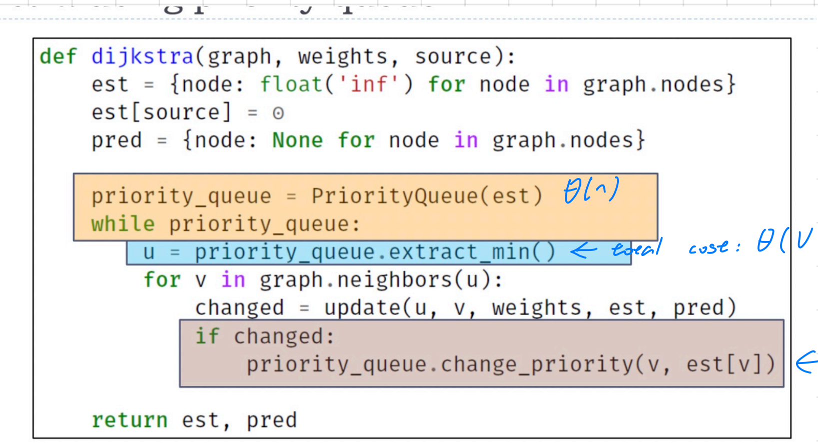
extract min: $\theta(\log n)$

change_priority: $\theta(\log n)$

initialization: $\theta(n)$.

⇓

Implemented using min-heap



```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)  θ(n)
    while priority_queue:
      u = priority_queue.extract_min() ← total cost: θ(V logV)
      for v in graph.neighbors(u):
          changed = update(u, v, weights, est, pred)
          if changed:
              priority_queue.change_priority(v, est[v])
    
    return est, pred
```

$\leftarrow \sum \deg(v) = \theta(E)$.

total cost: $\theta(E \log V)$.

Complexity: $\theta((V+E) \lg V)$

Prim Algorithm

Trees: Undirected graph $G = (V, E)$ is a tree ⇔

· it is connected

· it is acyclic.

· If $T = (V, E)$ is a tree, then $|E| = |V| - 1$.

Remark:

If $T=(V,E)$ is a tree,

- there is a unique path between any two nodes.
- adding any other edge e to T will create a unique cycle containing e.
- removing an edge will disconnect it.

A spanning tree of G is any graph $T=(V,E'\subseteq E)$ that is a tree,
for undirected graph $G=(V,E)$.

↓

Contains the smallest number of edges in E to connect all nodes in G.

Weight of spanning tree T of a weighted graph is

- total weight of all edges in T, $w(T)=\sum_{e\in T} w(e)$.

Minimum spanning tree (MST) is a spanning tree with smallest weight.

- may not be unique
- all MST for a given graph have same # of edges.

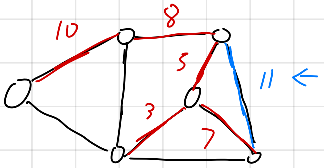
Problem:

input: a weighted undirected graph G.

output: the set of edges in MST of G.

Property: Given a MST T of $G=(V,E)$, let $e\in E$ be any edge in E but not in T:

⇒
- there is a unique cycle C containing e in $T\cup\{e\}$.
- e has the largest weight among all edges in cycle C.



create a unique cycle,
has the largest weight.

Greedy Algorithm: Prim

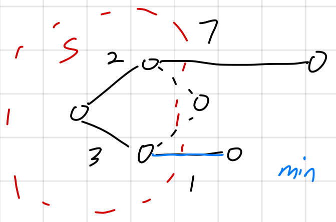
idea: - incrementally grow a partial tree $T(S)\subseteq E$ connecting a subset of nodes $S \subset V$.

- At beginning, $T(S)$ is a sub-tree of some MST of G
- At each iteration, grow $T(S')$ to include $S' = S \cup \{u\}$.

  s.t $T(S')$ still a sub-tree of MST,

  new node is reached via a greedy choice of a crossing-edge.

  the greedy choice is the min weight edge connect some node in S to some node in $U = V - S$.



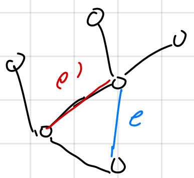
min weight outside S

MST Theorem: Let T be a sub-tree of a MST. If e is a min weight edge connecting T to some vertex not in T, then $T \cup \{e\}$ is also a subtree of MST.

↓

Loop invariant: when each iteration grow the subtree, new tree is still subtree of MST.

When all nodes are connected, we get MST.

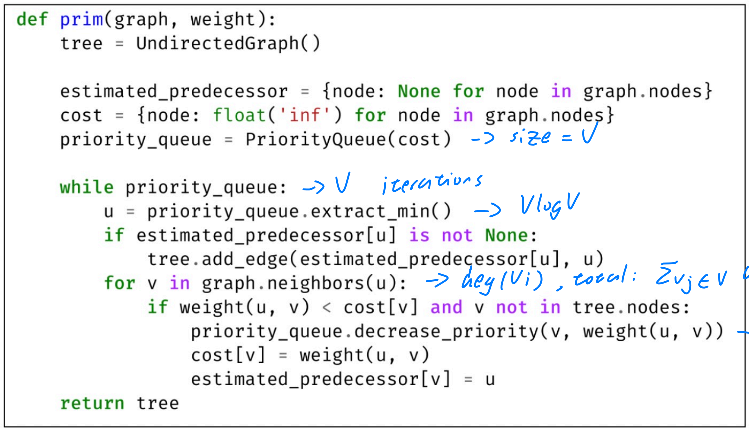


Implementation:

- Storing cost at node: each unvisited nodes v in U maintain $v.cost$, which is the smallest weight of any edge from v to visited nodes in S.

  ↓

  Priority Queue.



```
def prim(graph, weight):
    tree = UndirectedGraph()

    estimated_predecessor = {node: None for node in graph.nodes}
    cost = {node: float('inf') for node in graph.nodes}
    priority_queue = PriorityQueue(cost)

    while priority_queue:
        u = priority_queue.extract_min()
        if estimated_predecessor[u] is not None:
            tree.add_edge(estimated_predecessor[u], u)
        for v in graph.neighbors(u):
            if weight(u, v) < cost[v] and v not in tree.nodes:
                priority_queue.decrease_priority(v, weight(u, v))
                cost[v] = weight(u, v)
                estimated_predecessor[v] = u
    return tree
```

-> size = V

-> V iterations

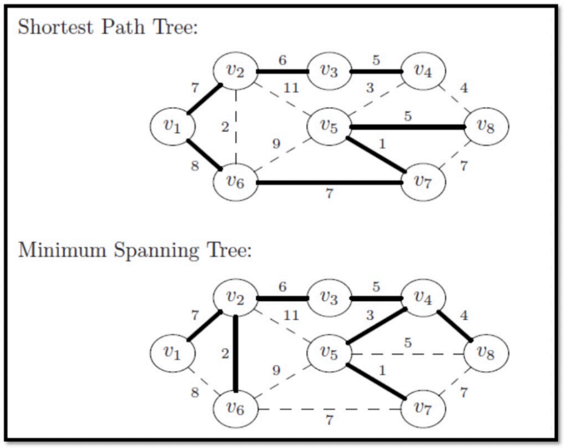
-> V log V

-> deg(Vi), total: $\sum_{v_j \in V} deg(v_j) = 2E$

-> E log V.

Complexity: $\Theta((V+E)\log V)$.

Comparison with Dijkstra



Shortest Path Tree:

Minimum Spanning Tree:

Dijkstra:

each node maintain best distance from source to current node.

• when inspecting a new edge (u,v).

v.distance = min(v.distance, u.distance + weight(u,v)).

Prim:

each node (not visited) maintains the minimum weight of any edge to reach a visited-node.

· when inspecting a new edge (u,V).

v.cost = min(v.cost, weight(u,v)).

Kruskal

idea: add edges gradually in a greedy manner using smallest weights, while maintaining what we have so far does not have any cycles.

how?

by checking whether nodes u,v are already connected.

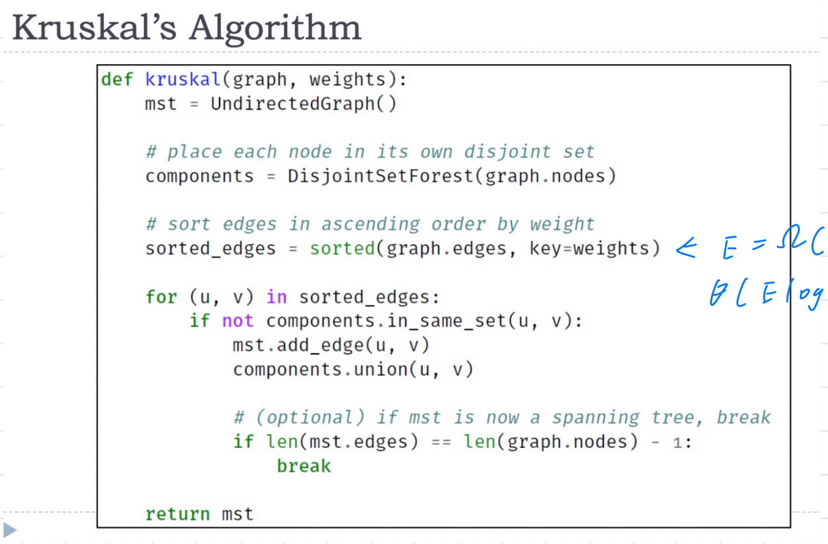
Disjoint Set Forest:

- represent a collection of disjoint sets over a set of elements.
  ex) $\{\{1,5,6\},\{2,3\},\{0\},\{4\}\}$
- Operation
  - union(x,y): union set containing x & y
  - in_same_set(x,y): return True/False if x & y are in the same set.

take $\Theta(\alpha(n))$ time, where n is # objects in the collection.

- $\alpha(\cdot)$: inverse Ackermann function:
  - grows very slowly
  - $\alpha(n) = O(\lg n)$.
  - Asymptotically, grow faster than $\Theta(1)$, in practice, $\rightsquigarrow \Theta(1)$.

- used to keep track of connected component of a dynamic graph. (CCs)
- Nodes of CCs are disjoint sets.
  - add edge (u,v) : .union(u,v).
  - check if u and v are connected : .in_same_set(u,v).



Kruskal's Algorithm

```
def kruskal(graph, weights):
    mst = UndirectedGraph()

    # place each node in its own disjoint set
    components = DisjointSetForest(graph.nodes)

    # sort edges in ascending order by weight
    sorted_edges = sorted(graph.edges, key=weights)

    for (u, v) in sorted_edges:
        if not components.in_same_set(u, v):
            mst.add_edge(u, v)
            components.union(u, v)

            # (optional) if mst is now a spanning tree, break
            if len(mst.edges) == len(graph.nodes) - 1:
                break

    return mst
```

$\leftarrow E = \Omega(V)$

$\Theta(E \log E) = \Theta(E \log V)$.

if graph disconnected, algorithm produces minimum spanning forest.

Kruskal vs. Prim:

Prim:

Binary heap : $\Theta(V \lg V + E \lg V)$ $(= \Theta(E \lg V))$ if graph connected).

Fibonacci heap: $\Theta(V \lg V + E)$.

Kruskal:

$\Theta(V + E \lg V)$ $(= \Theta(E \lg V))$ if graph is connected).

If graph is dense, prime with fibonacci heap is better.

In practice, kruskal may be faster for smaller dense graphs.

Clustering

- identify the groups in data
- loss minimization problem:

  assigning each data point a color so that the distance between closest pair is maximized.

- Distance Graph
  - given n data points $V = \{p_1 \cdots p_n\}$.
  - create a complete undirected graph $G = (V, E)$ s.t. for any $p_i \neq p_j$, there is an edge $(p_i, p_j) \in E$
  - the weight of an edge (p_i, p_j) is $w(p_i, p_j) = dist(p_i, p_j)$.

Clustering →
- create distance graph G.
- run either Prim's or kruskal to compute MST of G, T
- Delete largest edge in MST, obtain two components (clusters).

↓

We obtain k clusters for deleting k-1 edges in MST

Single-linkage-clustering (SLC)

- we can perform Kruskal, adding edges in ascending order of weights without forming cycles, and stop till we have a k number of components.

Complexity: $\Theta(E \lg V) = \Theta(V^2 \lg V)$ as $E = \Theta(V^2)$

problem: chaining-effect.

Complexity Theory

Many problems have brute force solution takes exponential time.

Polynomial Time:

- If an algorithm's worst case complexity is $O(n^k)$ for some k, it runs in polynomial time.

- Any polynomial is much faster than exponential for big n.

Not every problem solved in polynomial time.

What problems can and can not be solved in polynomial time?

=> Complexity Theory.

Ex: Eulerian problem: polynomial algorithm, "easy".

Hamiltonian problem: no polynomial algorithm, "hard".

Reduction: "Convert" Hamiltonian problem into Long Path problem in polynomial time. We called this reduction.

↓

Problem A reduces to problem B means "we can solve A by solving B".

Best time for A ≤ best time for B + polynomial.

- If A reduces to B, we say B is at least as hard as A

$P \overset{?}{=} NP$:

- The set of decision problems that can be solved in polynomial time is called P.

- The set of decision problems with "hints" that can be verified in polynomial time is called NP.

  - all of today's problems are in NP.
  - all problems in P also in NP.

Is P = NP? → means that any problem given "hint" verified in polynomial time can be solved in polynomial time.

No one knows.

NP-completeness:

- Suppose $(x_1 \cdots x_n)$ are boolean.
- A 3-clause is a combination made by or-ing and negating three variable.

Given: m-clause over n boolean variables

Problem: Is there assignment of $x_1 \dots x_n$ which makes all clauses true simultaneously?

No polynomial algorithm but easy to verify.

Cook's Theorem:

- Every problem in NP is polynomial-time reducible to 3-SAT.

- Corollary:

  If 3-SAT is solvable in polynomial time, then all problems in NP are solvable in polynomial time.

A problem is NP-complete if

- it is in NP;
- every problem in NP is reducible to it;

Hard Optimization Problem:

NP-complete → decision problem → yes or no

NP-hard → optimization problem. → find the best