CS 180 Discussion 1A/F

Week 5

Haoxin Zheng

11/03/2023

Announcements

- Midterm
  - In class, 11/9, next Thursday, 8:00 ~10:00 AM. Please arrive earlier than 8AM.
  - We will give you ~7min to upload your answers to Gradescope, be prepared.
  - Midterm review will be on 11/7, two previous midterm exams
  - Please write your solutions heavily and clearly.

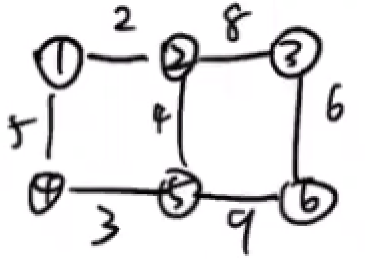
- Homework
  - HW#4 Q1 has been changed to **Exercise 11 on page 193**
  - Regrade request of HW2 will be open until 11/4 11:59PM.
  - The grading of HW3 will potentially be released before 11/8 (next Wednesday)

Minimum Spanning Tree (MST)

- Definitions:
  - A *minimum spanning tree* is the tree with minimum total weight among all trees given a positive weighted graph G.

- How to find a MST?
  - Run Prim’s/Kruskal’s algorithm

Minimum Spanning Tree - Kruskal



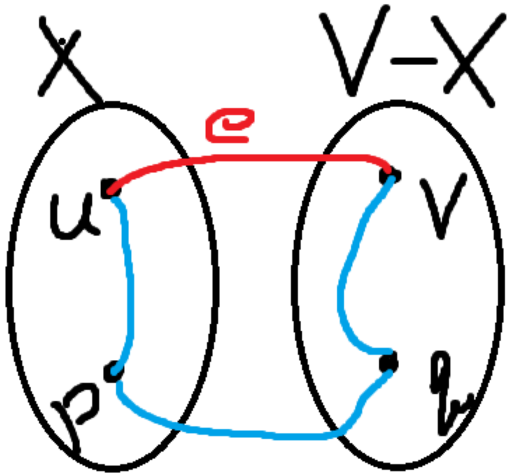
- **Given:** A connected undirected graph, G=(V, E), with edge length l(e)>0
- **Goal:** Find a set of edges $T^* \subseteq E, s.t.$
  - 1) G' = (V, T^*) is connected
  - 2) Minimize total cost $\sum_{e \in T^*} l(e)$

node	1	2	3	4	5	6
which connected component	1	2	3	4	5	6
union(1,2)	1	1	3	4	5	6
union(4,5)	1	1	3	4	4	6
union(2,5)	1	1	3	1	1	6
union(3,6)	1	1	3	1	1	3
union(2,3)	1	1	1	1	1	1

- **Kruskal's algorithm**:
  - Initialization: T = Ø (MST)
  - Sort edges s.t. $e_1 \leq e_2 \leq e_3 \ldots \leq e_m$
  - For i = 1,2, …, m:
    - (denote e_i as (u, v)).
    - If u, v are in different connected component
      - T = T + $\{e_i\}$
    - Stop when n-1 edges in T

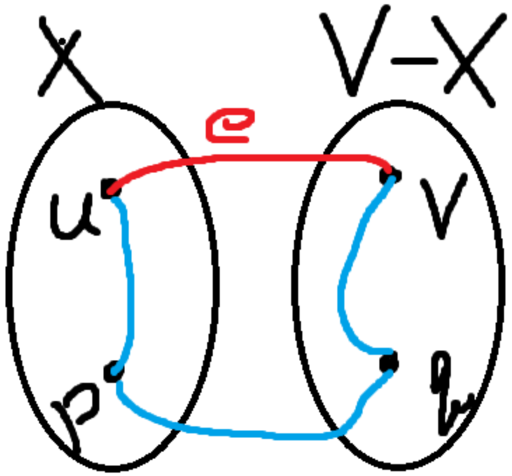
Proof of correctness for the Prim's algorithm

- Define "cut" of two node sets cut(A, B): All (u, v)∈E s.t. u∈ A, v∈ B
- Then the algorithm adds the min-cost edge in cut (X, V-X) in each iteration
- **Cut Property:** if edge e is the min-cost edge in cut(X, V-X) for any node set X, then e must be in the MST.
- **Proof:** Proof by contradiction.
  - Assume MST T^*, e(=(u, v)) is the min-cost edge of cut (X, V-X) but e ∉ MST T^*
  - In T^*, u, v are connected by another path. Then we know u ----> p -> q ----> v.
  - p: final node of X, q: first node of V-X of this u---------->v path
  - Define another tree T'=T^*-(p, q)+(u, v)
  - $T'$is still a connected graph since we have u ----> p -> q ----> v
  - Since l(e) = l(u, v) < l(p, q), we know l(T')<l(T^*).
  - Contradictive to the statement T^* is a MST
  - Proved.



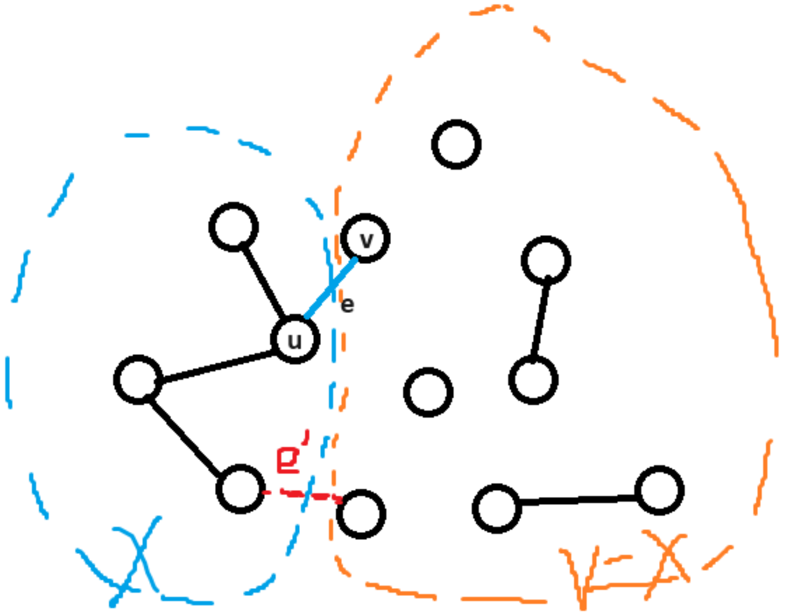
Proof of correctness for the Kruskal's algorithm

- Define "cut" of two node sets cut(A, B): All (u, v)∈E s.t. u∈ A, v∈ B
- Then the algorithm adds the min-cost edge in cut (X, V-X) in each iteration
- **Cut Property:** if edge e is the min-cost edge in cut(X, V-X) for any node set X, then e must be in the MST.
- **Proof:** Proof by contradiction.
  - Assume MST T^*, e(=(u, v)) is the min-cost edge of cut (X, V-X) but e ∉ MST T^*
  - In T^*, u, v are connected by another path. Then we know u ----> p -> q ----> v.
  - p: final node of X, q: first node of V-X of this u---------->v path
  - Define another tree T'=T^*-(p, q)+(u, v)
  - $T'$is still a connected graph since we have u ----> p -> q ----> v
  - Since l(e) = l(u, v) < l(p, q), we know l(T')<l(T^*).
  - Contradictive to the statement T^* is a MST
  - Proved.



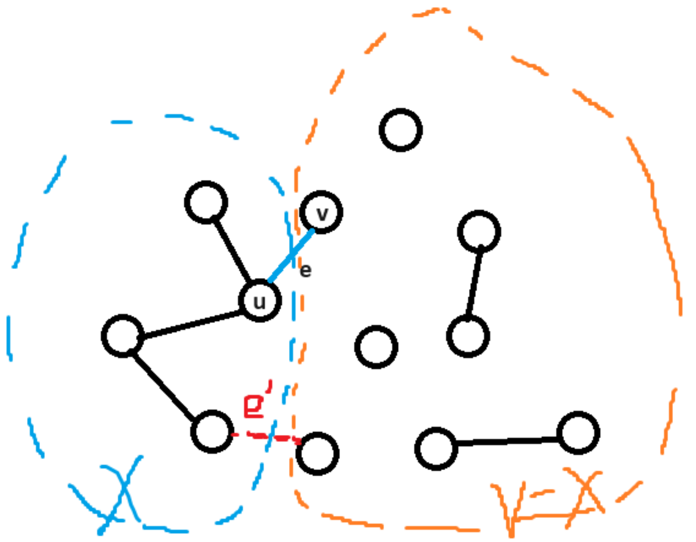
Proof of correctness for the Kruskal's algorithm

- The only difference is how to build the set X.
- Following the sorting order, we will pick the edge $e = (u, v)$ if when u, v are in different connected component.
- We can let the set X be the connected components of u (or v)
- Then, the edge $e = (u, v)$ is the min among all cut edges (min-cut). Why?



Proof of correctness for the Kruskal's algorithm

- The only difference is how to build the set X.
- Following the sorting order, we will pick the edge $e = (u, v)$ if when u, v are in different connected component.
- We can let the set X be the connected components of u (or v)
- Then, the edge $e = (u, v)$ is the min among all cut edges (min-cut). Why?
  - Proof by contradiction.
  - Assume e is not the min-cut, but another cut e' is.
  - This means, $l_{e'} < l_e$. However, by how we picking edges we know e' should have been picked already.
  - Therefore, e' belongs to the connected component of u,
  - It should be part of the X but not the cut between X & V-X
  - Contradiction



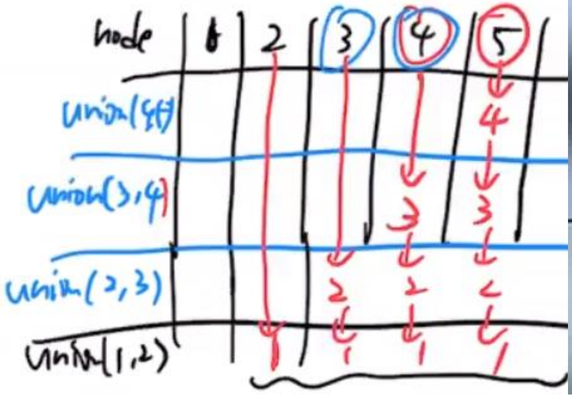
Union-Find Structure - Array

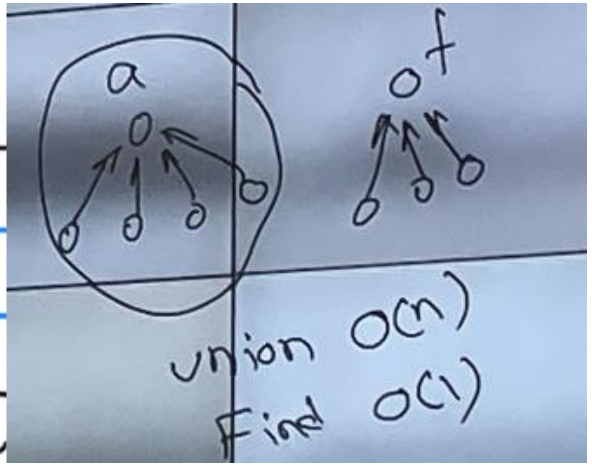
- How to check u, v are in the same component? – using union-find structure.
- Union-Find:
  - Store n elements and their sets
  - Union (u, v):merge set of u's and v's
  - Find (u): return the set "name" of u
    - Example: If Find(u) = Find(v) <=> u and v are in same sets

- **Kruskal's algorithm**:
- Initialization: T = Ø (MST)
- Sort edges s.t. $e_1 \leq e_2 \leq e_3 \ ... \leq e_m$
- For i = 1,2, …, m:
  - (denote e_i as (u, v)).
  - **If u, v are in different connected component**
    - T = T + $\{e_i\}$
  - Stop when n-1 edges in T

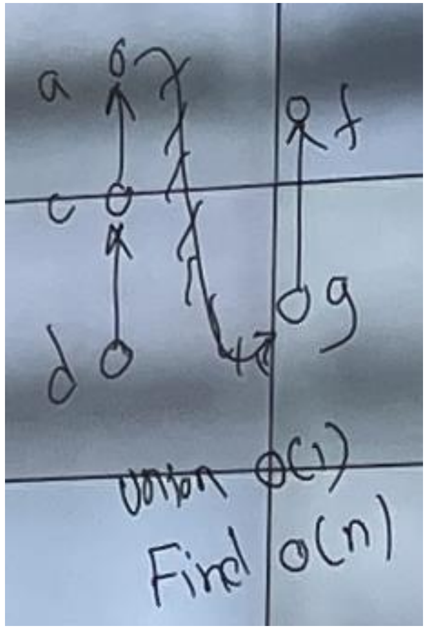
Union-Find Structure - Array

- Array/One-layer Tree implementation
  - Find: O(1)
  - Union: O(n)



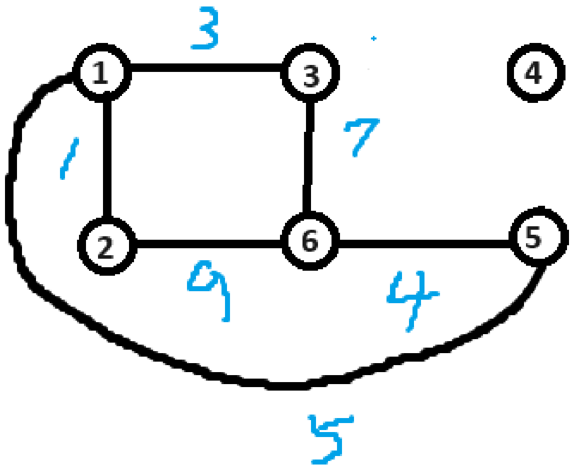


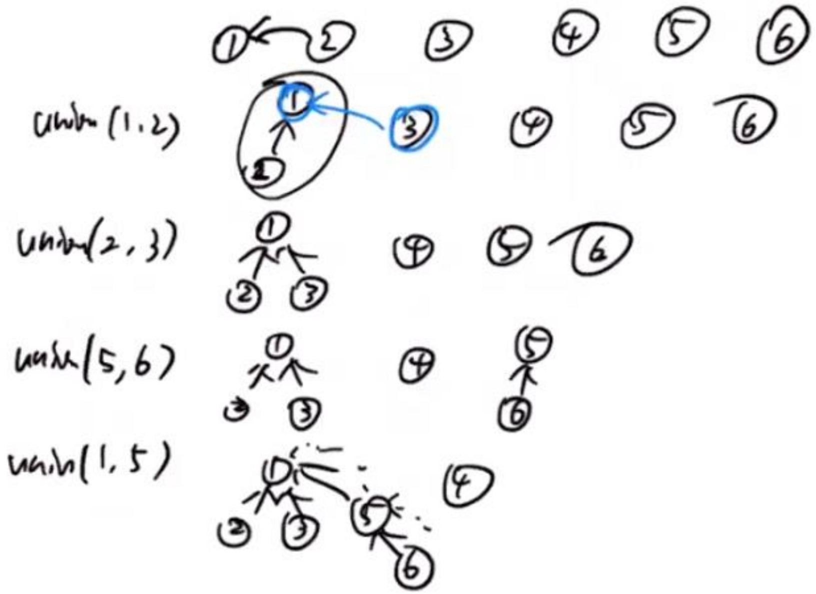
- Linkedlist implementation:
  - Find: O(n)
  - Union:O(1)



Union-Find Structure – Tree-based

- **Root**: Name of the set
- **Find**(u): traverse to the root, find the name
- **Union**(u, v): merge the smaller tree to the larger tree
- **Time Complexity**:
  - Find(u): O($\log n$) since tree's depth$\leq \log n$
  - Union(u, v): O($\log n$)
    - Find (u's root) ----- O(log n)
    - Find (v's root) ----- O(log n)
    - Point u's root to v's root
  - Applied to Kruskal's algorithm:
    - O(m) times find -> O(mlogn)
    - One sort -> O(mlogm)
    - In total: O(mlogm + mlogn) (*Write O(mlogm) is good enough*)





Divide and Conquer

- A **divide and conquer algorithm** is a strategy of solving a large problem by
  - breaking the problem into smaller sub-problems
  - solving the sub-problems, and
  - combining them to get the desired output

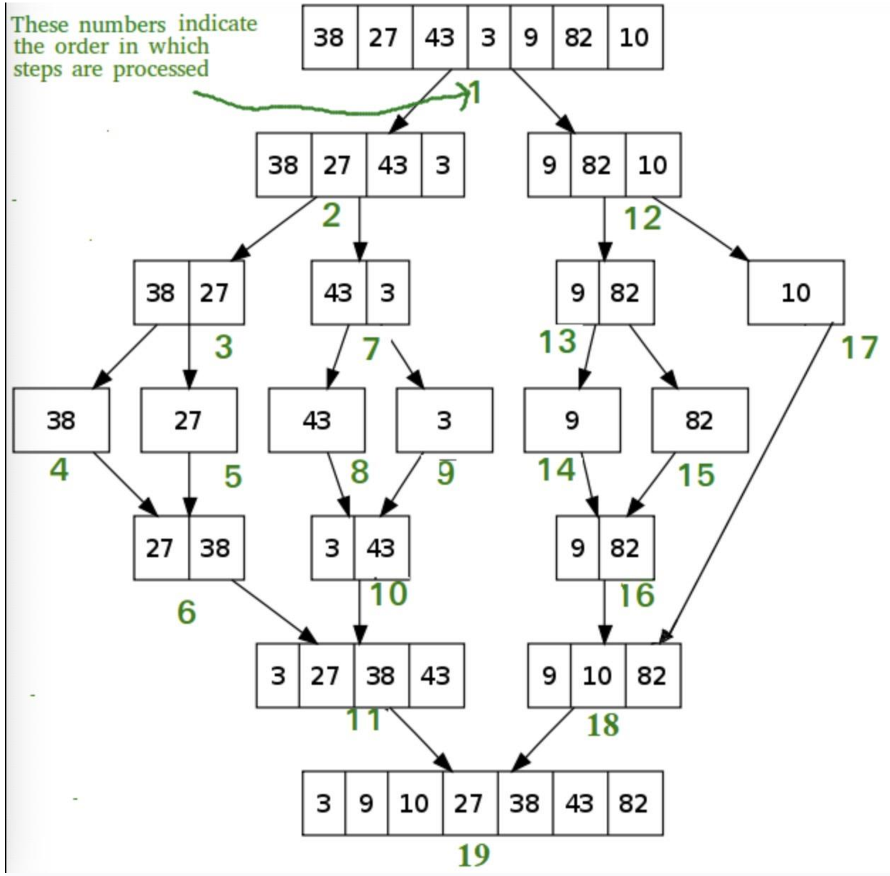
- Example: Binary search in divide and conquer problem:
  - Think of asking you to find an empty bottle along with n-1 bottles of waters with same weight, and only give you a balance to use.
  - $T(n) = T\left(\frac{n}{2}\right) + C \rightarrow T(n) = O(logn)$

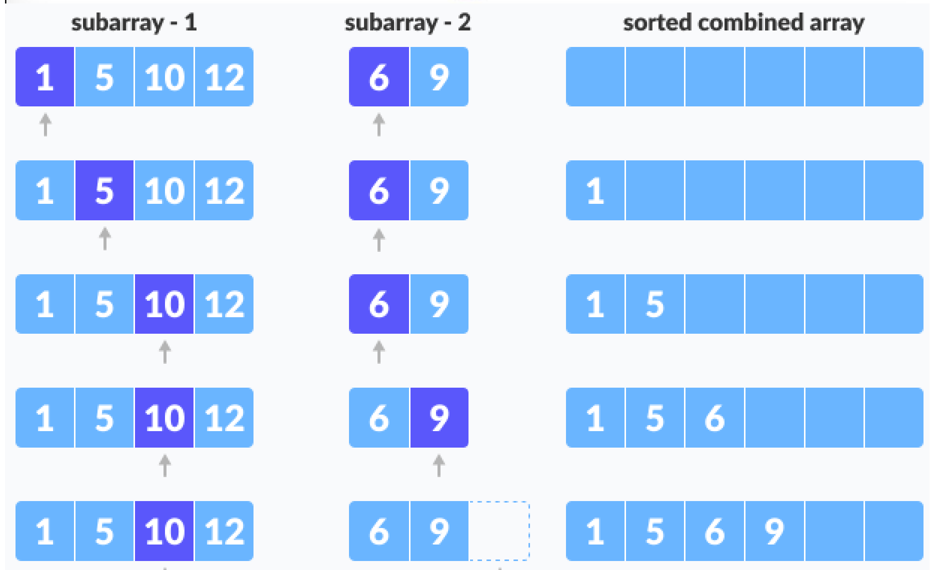
Merge Sort

- Given a list of integers
- Sort them in non-decreasing order
- Time Complexity: O(nlogn)

```
MergeSort(A, l, r):
-     if l > r
-         return
-     m = (l+r)/2
-     mergeSort(A, l, m)
-     mergeSort(A, m+1, r)
-     merge(A, l, r)

MergeSort(A, 0, length(A)-1)
```





Algorithm: Partition + Merge

- Partition the list into two roughly equal (± 1) subsets
- Keep partition each subset into two roughly equal (± 1) subsets, until only 1 element left in each partition
- Merge from small pieces back to the list with original length

Algorithm: Time Complexity

- $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right)$ + merge(L,R)
- Time complexity of merge(L,R) = Cn
- $T(1) = O(1)$
- $T(n) = 2T\left(\frac{n}{2}\right) + Cn$

$$= 2[2T\left(\frac{n}{4}\right) + C(\frac{n}{2})] + Cn$$

$$= 2^2 * T(\frac{n}{2^2}) + 2Cn$$

$$= 2^3 * T(\frac{n}{2^3}) + 3Cn$$

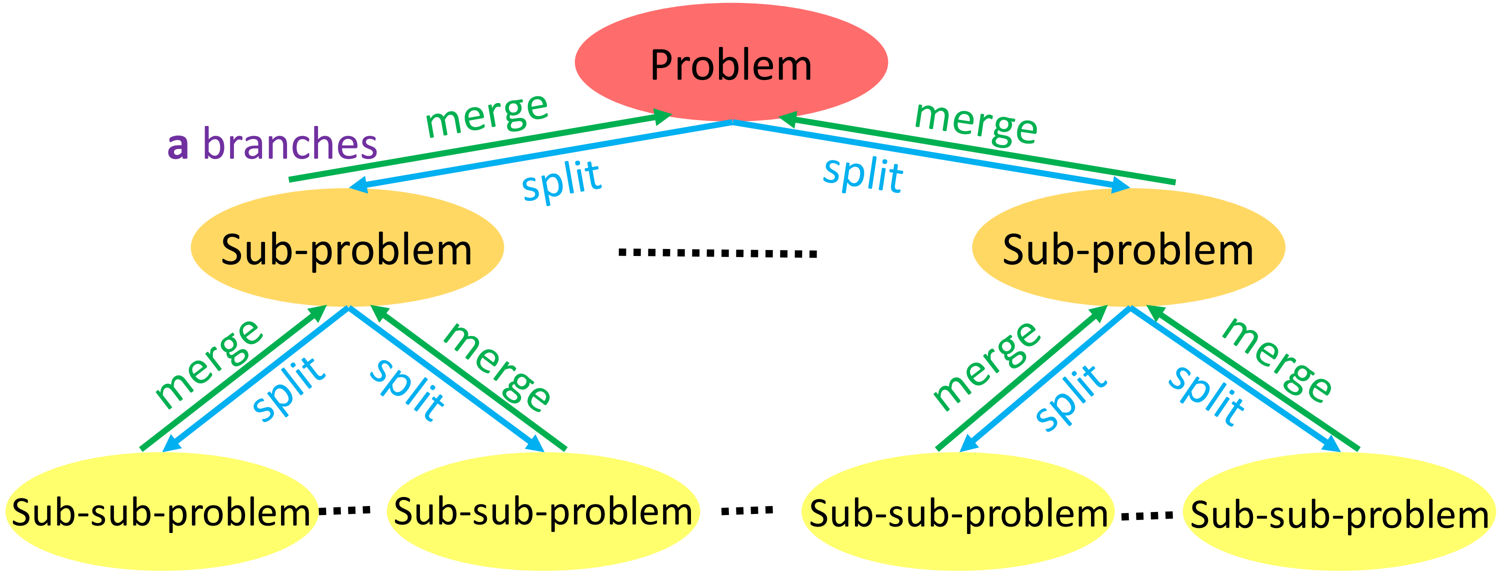
......

$$= 2^w * T(\frac{n}{2^w}) + wCn$$

- $\because \frac{n}{2^w} = 1 , \therefore w = logn$
- $\therefore T(n) = n + logn * Cn = O(nlogn)$

Divide and Conquer – Time Complexity

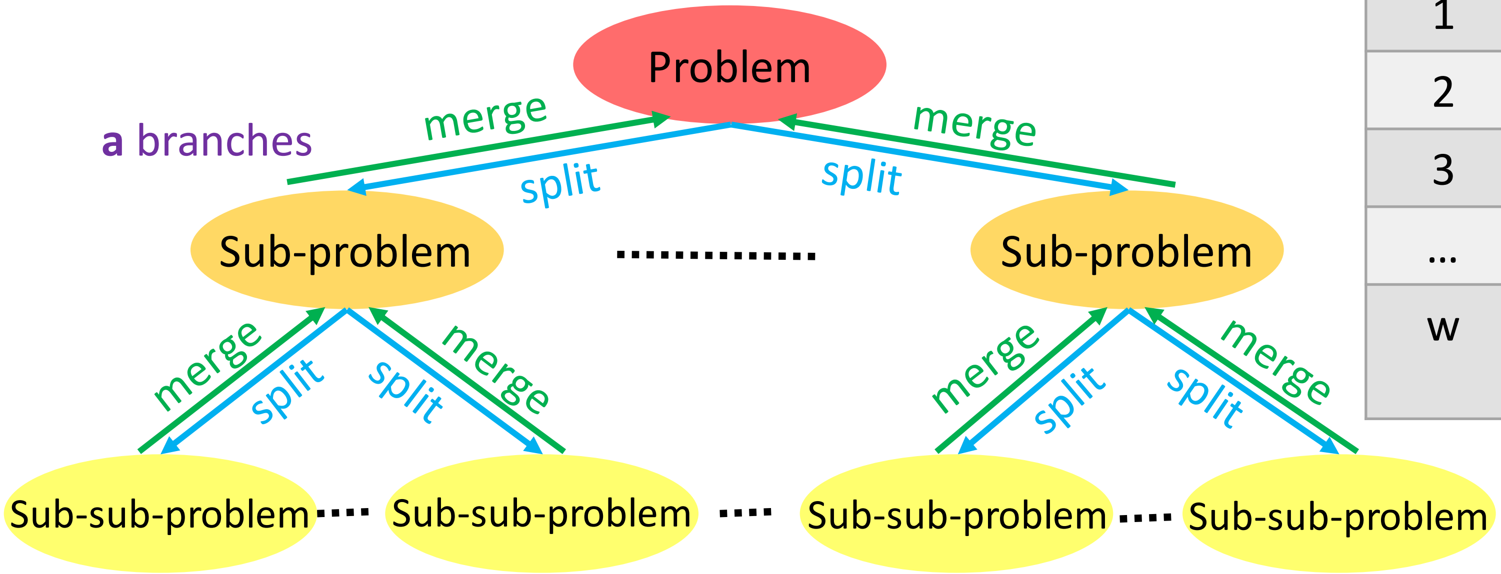
- We split the problem of size $\boldsymbol{n}$ into $\boldsymbol{a}$ subproblems with size of $\boldsymbol{n/b}$
- Time complexity of merging subproblems with size of $\boldsymbol{n/b}$ is $\mathrm{C}\boldsymbol{n^k}$



$$\boldsymbol{T(n) = aT(n/b) + cn^k}$$

Divide and Conquer – Time Complexity

- We split the problem of size $\boldsymbol{n}$ into $\boldsymbol{a}$ subproblems with size of $\boldsymbol{n/b}$
- Time complexity of merging subproblems with size of $\boldsymbol{n/b}$ is $\mathrm{C}\boldsymbol{n^k}$



$$\boldsymbol{T(n) = aT(n/b)+cn^k}$$

layer	# of sub-pb	Size of sub-pb	Cost of merging
1	1	n	cn^k
2	a	n/b	$a \times c(n/b)^k$
3	a^2	n/b^2	$a^2 \times c(n/b^2)^k$
…	…	…	…
w	a^w	$1\ or\ 2\ or\ some\ constant$	$a^w \times c(n/b^w)^k$

$$\boldsymbol{w \approx \log_b n}$$

Divide and Conquer – Time Complexity

- We split the problem of size $\boldsymbol{n}$ into $\boldsymbol{a}$ subproblems with size of $\boldsymbol{n/b}$
- Time complexity of merging subproblems with size of $\boldsymbol{n/b}$ is $\mathrm{C}\boldsymbol{n^k}$

- $T(n) = aT(n/b) + cn^k$

$$= a^2T(n/b^2) + ac(n/b^2)^k + cn^k$$

$$= \dots$$

$$= cn^k(1 + \frac{a}{b^k} + \left(\frac{a}{b^k}\right)^2 + \cdots + \left(\frac{a}{b^k}\right)^w)$$

layer	# of sub-pb	Size of sub-pb	Cost of merging
1	1	n	cn^k
2	a	n/b	$a \times c(n/b)^k$
3	a^2	n/b^2	$a^2 \times c(n/b^2)^k$
…	…	…	…
w	a^w	$1\ or\ 2\ or\ some\ constant$	$a^w \times c(n/b^w)^k$

$$\boldsymbol{w \approx \log_b n}$$

Divide and Conquer – Time Complexity

- $T(n) = aT(n/b) + cn^k$

$$= a^2T(n/b^2) + ac(n/b^2)^k + cn^k$$

$$= \dots$$

$$= cn^k\left(1 + \frac{a}{b^k} + \left(\frac{a}{b^k}\right)^2 + \cdots + \left(\frac{a}{b^k}\right)^{\log_b n}\right)$$

- So that we have:
  - $\frac{a}{b^k} == 1$: Time complexity $T(n) = O(n^k \log_b n)$ <- MergeSort's
  - $\frac{a}{b^k} > 1$: Time complexity $T(n) = O(n^{\log_b a})$ (after some math)
  - $\frac{a}{b^k} < 1$: Time complexity $T(n) = O(n^k)$ (after some math)

Count Inversions

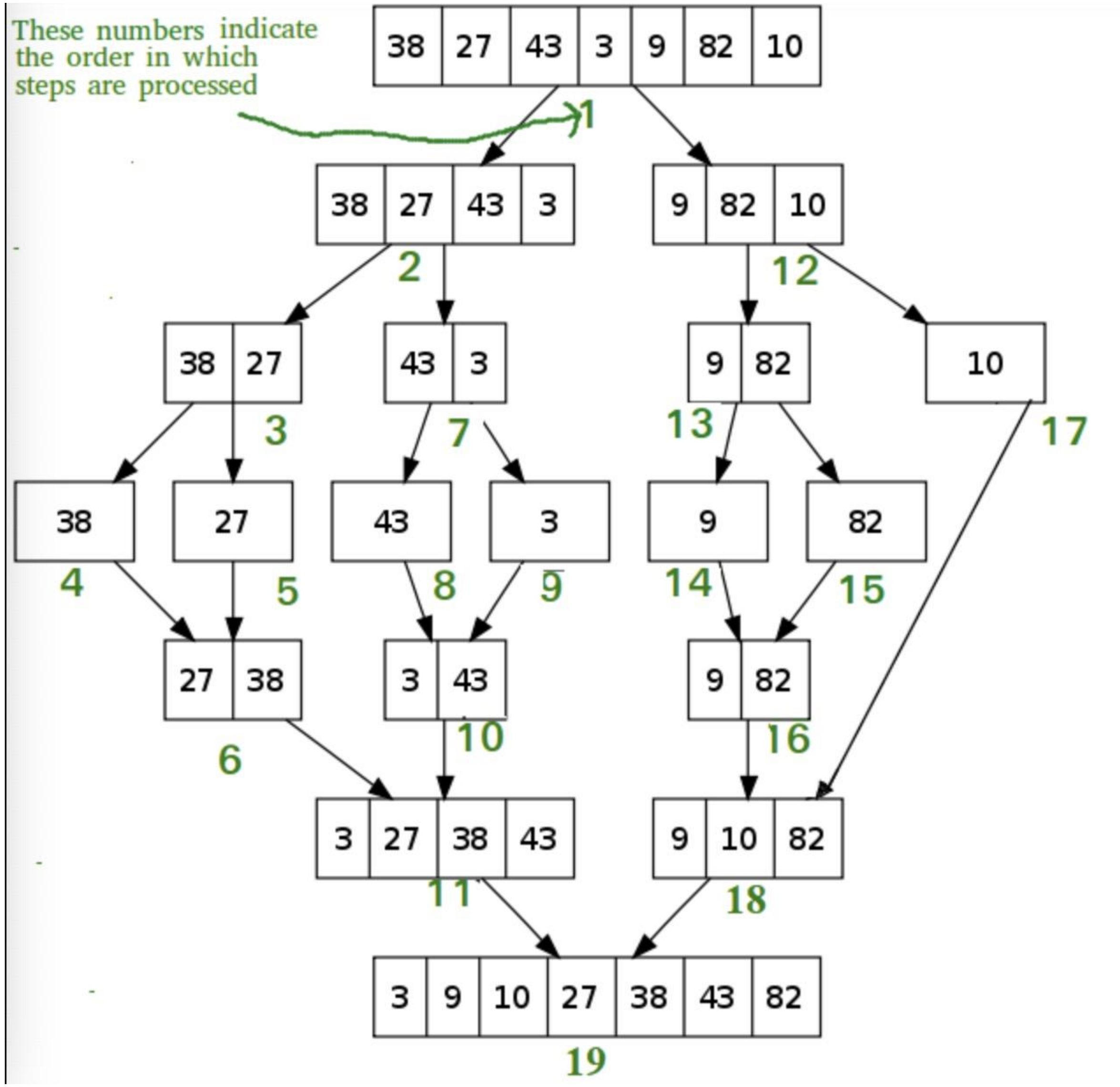
- Problem definition:
  - Assume we consider the ascending order as the normal order.
  - Given an unsorted list of unique numbers, count the number of inverted pairs.
  - Inverted: for pair $(n_i, n_j), i < j \ and \ n_i > n_j$

- Example: [2, 0, 3, 1]. Number of inverted pairs are 3:
  - (2, 0), (2, 1), (3, 1)

Count Inversions

- High-level divide-and-conquer algorithm:
  - Follow the idea of merge sort, sort the list to be in ascending order. During this step, we count the inverted pairs step by step.
  - 1) We keep a counter to count how many inverted pairs.
  - 2) Split the entire list into partitions with the smallest size (here is 1)
  - 3) During each merging operation, we will have a lhs sub-array and rhs sub-array, and merge these two arrays using the two-pointers approach taught before.
  - 4) In each step 3), if a rhs element e need to be pushed into the combined array while there are m elements remaining in the lhs sub-array, then this means these m elements are all larger than the e. This means there appear m inverted pairs (recall the definition of inverted pairs).
  - 5) We do counter+=m.
  - 6) After merge sort, we return counter.
- $T(n) = 2T\left(\frac{n}{2}\right) + Cn, T(1) = 1 \rightarrow O(nlogn)$

Count Inversions



6	8	1	3	2	4	0

Count Inversions

- Prove by induction.
- Base case:$i = 1$. An array of length 1 is automatically sorted and has 0 inversions.
- Assume $i = k$, that if the algorithm correctly sorts and counts inversions in arrays of length 2^k
- Given an array of length2^{k+1}. Each of the left half and right half is an array of length 2^k. By the inductive hypothesis, these two half arrays yields two sorted subarrays, $L\&R$, and correctly computes the number of inversions internal to each half. Thus, we now need to show that ***the merge step correctly counts the number of inversions between L and R*** and then we are done.
  - Proof by induction. We want to focus on the number of elements j that have been added into combined array A.
  - When $j = 1$, there is 1 element being added in combined array A. The algorithm can calculate the number of inversions between $L\ and\ R$ correctly with respect to this certain element, obviously.
  - Assume $j = t$, the algorithm can still calculate the number of inversions between $L\ and\ R$ correctly.
  - We want to prove when $j = t + 1$, *the merge step correctly counts the number of inversions between L and R* as well. This means we need to show the algorithm fully considers all the possible inversion pairs that contain the $(t + 1)^{th}$ smallest element we are currently adding to the array A.
  - Assume the element we are currently dealing with is x, it's the $(t + 1)^{th}$ smallest element to the array A. Two situations:
    - 1) x is from L. There is no element before x in L, which means there is no element in $L\&R$ is less than x. **0 inversions added.**
    - 2) x is from R. This means $x <$ all the elements remained in L. **# inversions += # of elements in *L*** by algorithm.
  - In both situations, we have looked through all possible inversion pairs that including x in $L\&R$ by the algorithm design.
  - Proved.
- Proved