Master Theorem

Theorem: If $T(n) = aT(\frac{n}{b}) + O(n^d)$, then we have

$$T(n) = \begin{cases} O(n^d), & a < b^d \\ O(n^{\log_b a}), & a > b^d \\ O(n^d \log n), & a = b^d \end{cases}$$

For each level, a problem is divided into a problems, for level k there are a^k problems. Level $\log_b n$ is the deepest one with $a^{\log_b n}$ problems. At this level the running time is $a \log_b n$. The total merge time for level k is $a^k \left(\frac{n}{b^k}\right)^d$.

By simplification, we have

$$T(n) = O(n^d) \cdot \left(1 + \frac{a}{b^d} + \cdots + \left(\frac{a}{b^d}\right)^{\log_b n}\right)$$

Discuss the relationship of a and b^d, and we can yield the above result. The three results also corresponds the cost of *merging* and *base cases*.

Median of Medians: we try to find a *good* pivot. There is a trade-off in the algorithm, we need to think about th time and quality of the pivot. So we turn to look at the recursive running time

$$T(n) = T(c \cdot n) + T_p + O(n)$$

- We partition S into subsets with size 5. [$O(n)$]
- Find the medians of them. [$O(n)$]
- Fix v to be median of those medians. This steps requires recursion and yields a time complexity of $T(n/5)$.
- It can be shown easily (using a figure) that v is no greater than $\frac{3}{10}$ integers or no less than $\frac{3}{10}$ integers, so we have

  $$T(n) = T(0.2n) + T(0.7n) + O(n)$$

  We can prove by induction that $T(n) \leq Bn$

  Base case: $T(1) = 1 \leq Bn$

  Inductive step:

  $$T(n) \leq T(0.2n) + T(0.7n) + Cn \leq 0.9Bn + Cn$$

  Set $B = 10C$ we yield $T(n) \leq 10Cn = O(n)$.

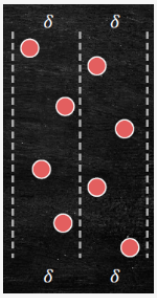
Closest Pair

> Input: a set of n points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
>
> Output: a pair of distinct points whose distance is smallest

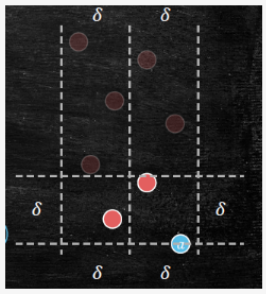
Apparently, the brute force approach has a time complexity of $O(n^2)$. We can improve it by sorting. For one-dimensional case, sorting the points by the x-coordinate has a time complexity of $O(n \log n)$. We try to extend the idea to more general cases.

- We first sort the points by the x-coordinate
- We divide it into two sides so that each side has $n/2$ points and repeat the process recursively

As we have seen previously, the combination process need to be elaborated, i.e. we need not compute all pairs. We denote δ_L, δ_R to be the smallest distance on left and right, $\delta = \min\{\delta_L, \delta_R\}$. We draw two lines with δ apart from the middle line and focus on the points between two lines. But in worst cases all the points might fall within the two lines, as shown in the following figure



Hence, the problem transfers to determining *what kind of pairs is impossible to be the closest one*. We fix on one point a and consider the pair (a, b) where $y_b > y_a$. Actually, we can focus on the $2\delta \times \delta$ rectangle (if b is outside the triangle, it's impossible that $d(a, b)$ takes on minimum values). In this case we can actually **bound** the number of points.



For the following part, we prove that the number of points in a $\delta \times \delta$ square is at most 4. We divide the square into four sub-squares. Since two points are at most $\frac{\delta}{2} < \delta$ apart, then there are at most one point in each square. Hence we can deduce that there are at most four points in a square.

Hence, the optimized approach is listed as follows

- We focus on a point a and a pair (a, b), b is above a
- We only need to compute *seven* b above a.

Now, we try to compute the time complexity of the algorithm

- Divide: $O(n \log n)$
- Recurse: $2T(0.5n)$
- Combine: $O(n \log n)$

Hence, we have $T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$, it's claimed that $T(n) = O(n \log^2 n)$ [HW, try to prove by induction]

Note that we can further optimize the algorithm by sorting in the beginning

- Sorting by the x-coordinate at first enables us to get all the separation points P used for recursion.
- Then we try to sort by y-coordinate, note that we only need P for separation, so the points can be sorted by y. Through recursion we can show that the array is always sorted in terms of y.
- Based on this observation, the time complexity can be reduced to $O(n \log n)$

- Basic steps for FFT
  - Choose $2d - 1$ distinct numbers $x_0, \ldots, x_{2d-2}$ and computer the values of $p(x_i)$ $q(x_i)$ respectively
  - For each $i = 0, 1, \ldots, 2d - 2$, we can compute $r(x_i) = p(x_i)q(x_i)$, so we obtain interpolation for $r(x)$. Note that actually $r(x)$ is already certain, but we don't know the accurate coefficients.
  - Finally, we recover $(c_0, \ldots, c_{2d-2})$, the polynomial $r(x) = \sum_{i=0}^{2d-2} c_i x^i$ from the second step.

We apply **Even-odd Decomposition**, $p(x) = p_e(x^2) + x \cdot p_o(x^2)$. Choose x_1 and x_2 such that $x_1 = -x_2$, we have $p_e(x_1^2) = p_e(x_2^2)$ and $p_o(x_1^2) = p_o(x_2^2)$. Based on this property, we know that we can decompose *two size-*D computations into *two size-*$\frac{D}{2}$ computations. In this case, we have
$T(D) = 2T(\frac{D}{2}) + O(D) \Rightarrow T(D) = O(D \log D)$.

But we are not done yet, as we observe that the x on the second level are all squares. If we want to continue our idea, we must introduce complex numbers. Suppose we fix D, we know that D numbers are $\omega^0, \ldots, \omega^{D-1}$, where $\omega = e^{\frac{2\pi}{D}i}$. We only need one parameter ω to represent to D numbers. We can further use ω^2 to represent the next level numbers. Putting everything together, we yield the following steps.

- If $\omega = 1$, return $p(1)$.
- Recursively use $FFT(p_e, \omega^2)$ to get $[p_e(\omega^0), \ldots, p_e(\omega^{D-2})]$
- Recursively use $FFT(p_o, \omega^2)$ to get $[p_o(\omega^0), \ldots, p_o(\omega^{D-2})]$
- For $t = 0, 1, \ldots, D-1$, we have $p(\omega^t) = p_e(\omega^{2t}) + \omega^t \cdot p_o(\omega^{2t})$.
- Finally, we return $[p(\omega^0), \ldots p(\omega^{D-1})]$.

For multiplication, the time complexity is apparently $O(d)$

Recovery

We already have a set of $(\omega^i, r(\omega^i))$, where $\omega = e^{-\frac{2\pi}{D}i}$, which is written in the following form

$$\mathbf{r} = \mathbf{Ac}, \quad \mathbf{A} = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ 1 & \omega & \ldots & \omega^{D-1} \\ 1 & \omega^2 & \ldots & \omega^{2(D-1)} \\ \ldots & \ldots & \ldots & \ldots \\ 1 & \omega^{D-1} & \ldots & \omega^{(D-1)^2} \end{bmatrix}$$

Now we recall some concepts in linear algebra. Given two *complex vectors* $\mathbf{a}, \mathbf{b} \in \mathbb{C}^n$, we have

$$< \mathbf{a}, \mathbf{b} >= \sum_{j=1}^{n} \overline{a_j} \cdot b_j$$

$\mathbf{a}, \mathbf{b}$ are orthogonal if $< \mathbf{a}, \mathbf{b} >= 0$, and orthonormal if $< \mathbf{a}, \mathbf{a} >= 1$ and $< \mathbf{b}, \mathbf{b} >= 1$. A square matrix A is an orthonormal matrix if every pair of its columns is orthonormal. **If A is an orthonormal, then A is invertible and** $A^{-1} = A^*$, where A^* is a *conjugate transpose* of A, defined as $(A^*)_{i,j} = \overline{A_{j,i}}$.

Now, we try to see whether $A(\omega)$ is orthonormal. Note that

$$< c_i, c_j >= 1 + \omega + \omega^2 + \cdots + \omega^{D-1} = 0$$

$$< c_i, c_i >= 1 + 1 + \cdots + 1 = D$$

So A is not orthonormal, but we can *scale* it, i.e. $\frac{1}{\sqrt{D}} A(\omega)$ is orthonormal. Hence, we can yield $\mathbf{A}^{-1}$

$$\mathbf{A}^{-1} = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ 1 & \omega^{-1} & \ldots & \omega^{-(D-1)} \\ 1 & \omega^{-2} & \ldots & \omega^{-2(D-1)} \\ \ldots & \ldots & \ldots & \ldots \\ 1 & \omega^{-(D-1)} & \ldots & \omega^{-(D-1)^2} \end{bmatrix}$$

The naïve way also need $O(D^2)$ times, and we can try to improve it by following a similar step to the first part of our algorithm. We let $s(x) = r(1) + r(\omega) \cdot x + \cdots + r(\omega^{D-1}) \cdot x^{D-1}$. Basically, we want to calculate the values for $s(1), s(\omega^{-1}), \ldots, s(\omega^{1-D})$. Note that $(\omega^{-1}, \omega^{-2}, \ldots, \omega^{1-D})$ is just the same as $(\omega^1, \ldots, \omega^{D-1})$ with *a clockwise orientation*. Hence, we can simply apply $FFT(s, \omega^{-1})$, with a time complexity of $O(d \log d)$.

Hence, the overall time complexity for FFT would be $O(d \log d)$.

More info

In the previous analysis, the function FFT is applied to quickly determine the values for $p(\omega), p(\omega^2), \ldots$. The metho is valid because of the special selection of ω. Here we can apply a similar method to quickly determine the value for $s(1), s(\omega^{-1}), \ldots$. By the former analysis we already see that ω^{-1} is also fit for the function.

Topological order

Now, we try to improve the algorithm by DFS. We run DFS first and record the start *time* and finish *time*.

```
time = 0
Function explore(v)
        start[v] = time
        time ++
        marked[v] = true
        for each (u,v) in E
                if marked[u] = false
                        explore(u)
    finish[v] = time
    time ++
```

Note that g is a tail, we guess that those with the *earliest finish time* is the tail. Once g is removed, we can choose the vertex with *second finish time*, i.e. f. So we make a conjecture that we can **sort vertices by descending order of finish time**.

To prove this, we claim that no arc (u, v) if $t[v] > t[u]$. Through the method of contradiction, supposing that (u, v) exists.

- (u, v) cannot be a tree edge, because u must be in a subtree with root v, so $t[u] > t[v]$
- Similarly, we can prove that (u, v) cannot be a forward edge
- (u, v) cannot be a cross edge, as a cross edge would require v to even start earlier than v.
- (u, v) cannot be a back edge (because of the DAG property, without which it is possible/certain that (u, v) is a back edge)

During DFS, when we finish a vertex, we can actually append it to the topological order (as it follows the order of finish time). Hence, the optimized time complexity is $O(V + |E|)$.

ssc

- DFS G^R and maintain a **sorted list** by the finish time
- DFS G by the descending order of the finish time, and we mark the vertices that we have reached.
- Each explore operation forms a SCC.

The time complexity for this algorithm is $O(|V| + |E|)$, as it basically runs two DFS (the second one in a given order). But we need to prove its correctness, as we have assumed here the order of step 2 *is* actually the topological order corresponding to the original SCC graph. That is, we want to prove that **each start point we choose is in the head SCC among the remaining graph**.

dijkstra

Thus, we can formulate **the Dijkstra Algorithm**

- Initialize
  - $T = \{s\}$
  - $tdist[s] = 0$, $tdis[v] \leftarrow \infty$ for all $v \neq s$ and $tdist[v] \leftarrow w(s, v)$ for all $(s, v) \in E$.
- Explore
  - Find $v \in T$ with smallest $tdist[v]$
  - $T \leftarrow T + \{v\}$
- Update $tdist[u]$
  - $tdist[u] = \min\{tdist[u], tdist[v] + w(v, u)\}$ for all $(v, u) \in E$

Note that we need to find the smallest $tdist[v]$ for $|V|$ times, to do this efficiently, we need to use heaps. So the general time complexity would be $O[(|V| + |E|) \log |V|]$. But if we use Fibonacci Heaps, we can further reduce it to $O(|V| \log |V| + |E|)$

bf

Bellman-Ford

Note that if negative edges exists in the original graph, then the Dijkstra algorithm can fail. To deal with the problem, we introduce Bellman-Ford Algorithm. Dijkstra is very clever for it follows a *special* order. Each edge can be only used once in updating. Now the order is not true, so we can only be stupid.

```
Function bellman_ford(G, s)
        dist[s] = 0
        dist[x] = MAX_INT for other x in V
        while there exists dist[x] that is updated
                for each (u,v) in E
                        dist[v] = min(dist[v], dist[u] + dist(u,v))
```

For the correctness of Bellman-Ford, we first introduce a lemma.

Lemma 1: After k rounds, `dist[v]` is the shortest distance of k-edge-path.

Proof: We try to prove by induction. Suppose it is true for $k - 1$ rounds, consider a k-edge-path of v: $(s, u_1, \ldots, u_{k-1}, v)$. By hypothesis, we have $dist[u_{k-1}] \leq d(s, u_1, \ldots, u_{k-1})$. By Bellman-Ford, we have $dist[v] \leq d(s, u_1, \ldots, u_{k-1}) + d(u_{k-1}, v)$

Lemma 2: The shortest distance of all $|V|$-edge-path can not be shorter than the shortest distance of all $(|V| - 1)$-edge-path unless there is a negative cycle.

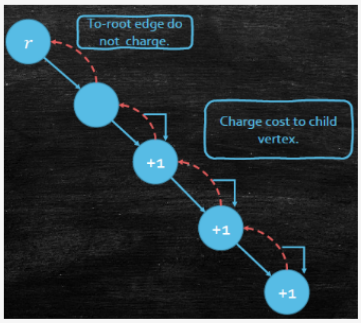
Proof: $|V|$-edge-path must contains a cycle. If the cycle is not negative, go through it do not make the distance smaller.

Hence, we can derive that after $|V| - 1$ rounds, $dist[v]$ is the shortest distance, otherwise G has a negative cycle. The time complexity would be $O(|V| \cdot |E|)$

Path Compression

Suppose we need to visit the vertices $v_1, \ldots, v_n$ before reaching s, then we attempt to move those vertices to the root node (without changing the relative position of the children of v_i). To analyze this intuition, we need to use amortized analysis again.

We attempt to prove that any m find operations totally cost $O(m \log^* n + n \log^* n)$, since the $n \log^* n$ cost does not increase by m, we can say the amortized cost of `find` is $O(\log^* n)$ for each operation (when $m >> n$). For th following analysis, we use full **charging**. The cost of this particular `find` is represented by the four red edges. The edge to root does not charge while all the others charge cost to all the child vertices.



The total cost of m thus finds consists of two parts

- Self-payment: $O(m)$
- Charging Cost: $\sum_m C(v)$ where $C(v)$ is the cost v has paid.

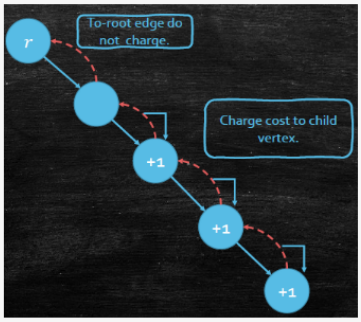
Recall the rank process described previously, note that a `rank=2` tree might have $h = 1$ due to path compression. Anyway, we keep the property `rank` which is quite important. Specifically, we update rank when we merge two trees, but we don't update any rank in path compression. It's easy to see that `rank[v] >= h[v]` for we only decrease tree's height.

Based on the above discussion, we can formalize the two lemmas.

Path Compression

Suppose we need to visit the vertices $v_1, \ldots, v_n$ before reaching s, then we attempt to move those vertices to the root node (without changing the relative position of the children of v_i). To analyze this intuition, we need to use amortized analysis again.

We attempt to prove that any m find operations totally cost $O(m \log^* n + n \log^* n)$, since the $n \log^* n$ cost does not increase by m, we can say the amortized cost of `find` is $O(\log^* n)$ for each operation (when $m >> n$). For th following analysis, we use full **charging**. The cost of this particular `find` is represented by the four red edges. The edge to root does not charge while all the others charge cost to all the child vertices.



The total cost of m thus finds consists of two parts

- Self-payment: $O(m)$

- Charging Cost: $\sum_m C(v)$ where $C(v)$ is the cost v has paid.

Recall the rank process described previously, note that a `rank=2` tree might have $h = 1$ due to path compression. Anyway, we keep the property `rank` which is quite important. Specifically, we update rank when we merge two trees, but we don't update any rank in path compression. It's easy to see that `rank[v] >= h[v]` for we only decrease tree's height.

Based on the above discussion, we can formalize the two lemmas.

Lemma 1. Parent's rank is strictly larger than the child.

Lemma 2. The tree of v has at least $2^{\text{rank}[v]}$ vertices.

We now group vertices by the final rank, requiring that group i has 2^{k_i-1} elements. With these groups, we can develop two kinds of charging: same group charging (SGC) and across group charging (AGC).

Lemma 3. We have at most $\log^* n$ groups.

Proof: The largest rank is at most $\log n$, and the last group is $[k = \log n, 2^k]$. We can see that there are $\log k$ group before group k. Hence, we have at most $\log^* n$ groups.

Lemma 4. Group $[k+1, 2^k]$ at most have $n/2^k$ vertices.

We first acknowledge the fact that the number of vertices of exactly rank k is at most $n/2^k$, as a vertex can only be used to create one rank k root. Hence, the number of vertices in this group is at most

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots + \frac{n}{2^{2^k}} \leq \frac{n}{2^k}$$

For SGC, and m `find` operations, suppose v have many SGCs denoted by $(u_1, v), \ldots$ It's clear that $u_1, u_2, \ldots$ is each SGC's parent. By path compression, we can assume that $u_{i+1} > u_i$ for all i. Every time we make SGC, we also do a path compression. Hence in this group there are at most $2^k - (k+1) < 2^k$ SGCs for v (i.e. all the nodes are associated with v in charging).

By Lemma 4, there are at most $n/2^k$ vertices in group k, so totally there are at most n SGC in a group. Therefore, the total cost of m `find` operations is $O(m \log^* n + n \log^* m)$.

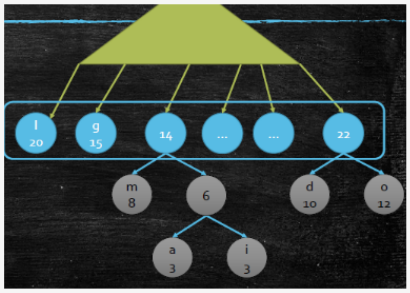
Huffman Encoding

Input: an alphabet A, a frequency function $f : A \to N$.

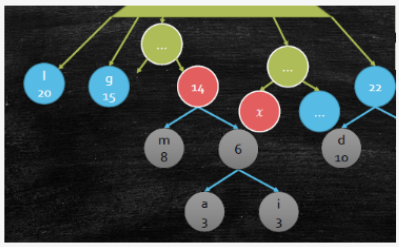
Output: a minimum cost prefix-tree for A.

We return to the big idea of proving greedy algorithms, *the local greedy choice do not ruin out OPT*. Assume we are still in a partial-OPT, after merging two smallest elements, prove we are still in a partial-OPT.

Suppose we are in an optimal partial tree T^*. We choose the smallest two nodes $u, v (f(u) \leq f(v))$ and merge them into a new node. We try to prove by contradiction, assuming that the two nodes u, v are not siblings.



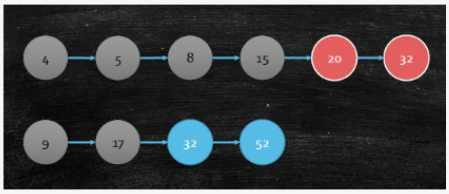
In the OPT, we check for the two lowest siblings. By the hypothesis we know that at least one of them are not u or v, denote the node by x. Here we know that the depth of u is smaller than the depth of x. Intuitively, we try to show that if we switch u and x, we can do build a better tree. We first show that the bottom *grey* part can be excluded from the graph. Switching the two blue nodes thus indeed make the costs lower.



We can sort the characters by their appearance using a heap, which yields a time complexity of $O(n \log n)$. Suppos the nodes are already sorted in the beginning, we can actually do better. The tasks we are facing are listed as follows

- Find two minimized appearance elements
- Delete two minimized element
- Insert a super node into the list

We can actually use priority queues to implement it, which only takes a time complexity of $O(n)$. Start from one linked list, we merge the first two points and create a new node to be the first node of the second linked list. We then continue to find the two smallest nodes and merge them, and continuously add the merged node into the second linked list. The basic process can be seen from the following figure.



- Case 1: we have 2^k characters with same frequency. Intuition is that the length of the code match the choices of the character. In this case, we apply information theory to yield its entropy

$$H(X) = 2^k \frac{1}{2^k} \log 2^k = k$$

- Case 2: we have n characters with frequency: $m \cdot 2^{-(n-1)}, m \cdot 2^{-(n-1)}, \ldots$. We can verify that the code length is $-\log p(a)$.

Makespan Minimization

> **Input:** m identical machines, n jobs with size p_i
>
> **Output:** the minimized max completion time (Makespan) of all these jobs on m machines.

We attempt to use **LDT** algorithm, and consider longest processing time first and insert jobs into the earliest finished machine. Assume we are in OPT, we put the longest pending job onto the earliest finished machine. But while we are trying to prove the correctness, we can construct a counter-example. The upper approach uses LDT and the lower one is the actual optimal.

Makespan minimization is a NP-hard problem. Find a poly time algorithm for it means $P = NP$. But we can at least show that greedy and LPT are better than arbitrary sequences. For greedy algorithm, we denote the last finished job as p_n. Then the actual workload before p_n is bigger than tm. Since $OPT \geq t$, by definition we have $ALG = t + p_n \leq 2OPT$. Hence, the approximation ratio is 2. We then give a counter-example to show that the approximation is indeed tight when m is large enough.

Then we consider LPT, we only consider the cases where p_n is the smallest job (if not, suppose p_j is smallest a attached to another machine, then we can remove it without influencing ALG).

k-centers

> **Input:** A metric space (V, d) and $k \in \mathbb{Z}^+$
>
> **Output:** A set of k centers, $S \subseteq V$ with $|S| = k$ that minimizes $f(S) = \max_{v \in V} d(S, v)$

Here $d(S, v) := \min_{s \in S} d(s, v)$ is the distance of v to its closest center $s \in S$, $f(S)$ is the maximum distance of any vertex $v \in V$ to its closest center.

A natural idea is to iteratively *pick the center of farthest to the existing centers*, i.e. maximize the reduction in the cost at each iteration. But the idea has problems, with an easy counter-example for $k = 2$, supposing that the vertices form a linked list. It is impossible for the algorithm to find optimal solutions at two quarter-points. The example also illustrates that the approximation ratio is at least 2. **We will show that it is indeed a 2-Approximation.**

Let $O = \{o_1, \ldots, o_k\}$ be the optimal solution with cost OPT, and $A = \{a_1, \ldots, a_k\}$ be the algorithm's output with cost ALG. We attempt to show that $ALG \leq 2 \cdot OPT$. For each o_i, let X_i be the set of vertices closest to o_i. And we have that $\{X_1, \ldots, X_k\}$ forms a partition of V. Consider the following two cases

- $A \cap X_i \neq \emptyset$ for each i. Then each X_i contains a center in $A = \{a_1, \ldots, a_k\}$, assume $a_i \in X_i$ WLOG, then we consider $v \in V$ and let X_i be the cluster containing V. Recall that $OPT = \max_{v \in V} \min_{o_i \in O} d(o_i, v)$. Each vertex u in X_i is closest to o_i, so we have $d(o_i, u) \leq OPT$. Thus $d(o_i, v) \leq OPT$ and $d(o_i, a_i) \leq OPT$, and by triangle inequality we have $d(a_i, v) \leq 2 \cdot OPT$. Hence, we have

  $$d(A, v) \leq d(a_i, v) \leq 2 \cdot OPT$$

  Note that $d(A, v) = d(a_i, v)$ iff v is closet to a_i and else otherwise.

- $A \cap X_i = \emptyset$ for some i. In this case, some X_i contains two centers in A. Suppose $a_r, a_j \in X_i$, let a_r be the second center chosen in X_i. By our greedy choice, $ALG = d(A, a_r) \leq d(a_j, a_r)$ before a_r is chosen. Thus $ALG \leq d(a_j, a_r)$ when the algorithm ends. Additionally, we have

$$d(a_j, a_r) \leq d(a_j, o_i) + d(a_r, o_i) \leq 2 \cdot OPT$$

Hence, we have proved $ALG \leq 2 \cdot OPT$ for both cases. We will show that further optimization will be more difficu than *dominating set* problem.

Dominating Set and Inapproximability

Give an undirected graph $G = (V, E)$, **a dominating set** is a subset of vertices S such that for any $v \in V \backslash S$, there is a vertex $u \in S$ that is adjacent to v. Given a undirected graph $G(V, E)$ and an integer $k \in \mathbb{Z}^+$, decide if G contains a dominating set with size k. We state that (without proof, will return later) there is no polynomial time algorithm for dominating set.

We claim that if there exists a polynomial time $(2 - \varepsilon)$-approximation algorithm $\mathcal{A}$ for k-center, then there exists a polynomial time algorithm $\mathcal{A}'$ that solves Dominating Set.

$\mathcal{A}'$ does the followings

- Given a dominating set input (G, k), apply $\mathcal{A}$ on (G, k)
- G has a dominating set of size k iff the k-center solution of (G, k) by $\mathcal{A}$ has cost 1.

  We will prove the statement as follows. Suppose $\mathcal{A}(G, k) = 1$, by definition of k-centers, G has a dominating set of size k. If G has a dominating set of size $k < |V|$, then we have $OPT_{k\text{-center}} = 1$, then the algorithm wil give $\mathcal{A}(G, k) \leq 1 \cdot (2 - \varepsilon) = 1$. Hence, we have $\mathcal{A}(G, k) = 1$.

Local Search

In local search, we usually start with a solution (normally an arbitrary one) and improve it by *local updates* until no more update improves the objective. Note that it is similar to greedy in that it follows locally optimal moves. But greedy algorithm iteratively builds a solution in a greedy way, but local search starts from a solution and iteratively improve it in a greedy way. A typical example is *max-cut problem*.

> Given an undirected graph $G = (V, E)$, a **cut** is a partition $\{A, B\}$ of V. The value of the cut is defined by $c(A, B) = |E(A, B)|$.
>
> Give an undirected graph $G = (V, E)$, find a cut $\{A, B\}$ with maximum value.

We define a local search algorithm

- Start with any partition $\{A, B\}$.
- If moving a vertex u from A to B or from B to A increases $c(A, B)$, move it.
- Terminate until no such movement is possible.

We can analysis the time complexity as follows.

- Each update searches for at most $O(|V|)$ vertices
- For each vertex, decide if the update is beneficial takes at most $O(|E|)$ time, for *at least one edge is added to the cut number*.
- Total number of updates is at most $|E|$, as each update increases the cut size by at least 1.
- Overall, the time complexity is $O(|V||E|^2)$.

For each vertex u, at least $\frac{1}{2}\deg(u)$ incident edge are in the cut. Thus, we have

$$c(A, B) \geq \frac{1}{2}\sum_{u \in V}\frac{1}{2}\deg(u) = \frac{1}{2}|E| \text{ by Handshaking Theorem}$$

- $|E|$ is an obvious upper bound to OPT.
- Therefore, $c(A, B) > \frac{1}{2}|E| > \frac{1}{2}OPT$, and it is a 0.5-approximation.