1. Assume w.l.o.g. that $\mathrm{pred}(i) = -1$ initially for every $i \in [0, n-1]$. Define system potential

$$\Phi = \sum_{k \in [0, \sqrt{n}]} n - \mathrm{latest}(k)$$

where $\mathrm{latest}(k)$ is the largest index i that $\mathrm{pred}(i) = k$ if one exists, or 0 otherwise. Let Φ_i be the system potential after i operations are performed. Hence, $\Phi_0 = O(n\sqrt{n})$ and $0 \leq \Phi_n \leq \Phi_0$.

If the ith operation has the actual cost k, then the change of system potential $\Phi_i - \Phi_{i-1} = -k + O(1)$. Thus, the amortized cost $\hat{c}_i = k - k + O(1) = O(1)$. Consequently, the sum of actual cost is upper-bounded by

$$\Phi_0 - \Phi_n + \sum_{i \in [1,n]} \hat{c}_i = O(n\sqrt{n}).$$

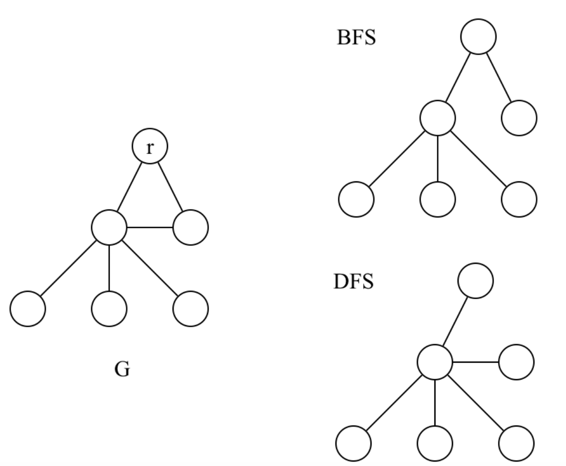
2. (a) As shown in Table 1, Fibonacci heaps is the best.

	arrays	binary heaps	Fibonacci heaps
n Insert()	$O(n)O(1) = O(n)$	$O(n)O(\log n) = O(n \log n)$	$O(n)O(1) = O(n)$
n Extract-Min()	$O(n)O(n) = O(n^2)$	$O(n)O(\log n) = O(n \log n)$	$O(n)O(\log n) = O(n \log n)$
$m = O(n \log n)$ Decrease-Key()	$O(m)O(1) = O(n \log n)$	$O(m)O(\log n) = O(n \log^2 n)$	$O(m)O(1) = O(n \log n)$
Total running time	$O(n^2)$	$O(n \log^2 n)$	$O(n \log n)$

Table 1: Running time of different implementations of Prim's algorithm.

(b) Let $\mathcal{H}$ be the collection of trees. Then, both binary heaps and Fibonacci heaps have the runtime $O(n \log n)$, faster than the runtime of the array implementation $O(n^2)$. Since we assume that binary heaps is easier to implement than Fibonacci heaps, binary heaps is the best for $\mathcal{H}$.

3. Yes, such a graph G exists, depicted as follows. The roots of BFS and DFS trees are both r.



4. (a) Replace Line 9 with **for** $k = 1$ to n **do**.

   (b) Algorithm 2 returns an incorrect answer when the nth node is an intermediate node in some shortest path. This bad case can be avoided if the degree of the nth node is 1. Therefore, one may set $\mathcal{H}$ as the collection of all connected graphs whose nth node has degree 1.

5. If such a subsequence exists, then Algorithm 1 returns "Yes" with probability $\geq \delta$ for some constant $\delta > 0$. If such a subsequence does not exist, then Algorithm 1 always returns "No."

$x \leftarrow$ an element sampled uniformly at random from $\{a_1, a_2, \ldots, a_n\}$
$y \leftarrow$ an element sampled uniformly at random from $\{a_1, a_2, \ldots, a_n\} \setminus \{x\}$
$z \leftarrow$ an element sampled uniformly at random from $\{a_1, a_2, \ldots, a_n\} \setminus \{x\}$
$\hat{d} \leftarrow \gcd(|x - y|, |x - z|)$
Use a linear scan to check the intersection between $a_1, a_2, \ldots, a_n$ and
$\quad \ldots, x - 2\hat{d}, x - \hat{d}, x, x + \hat{d}, x + 2\hat{d}, \ldots$
if *the intersection contains at least* $n/2$ *elements* **then**
$\quad$ output "Yes"
else
$\quad$ output "No"
end

Algorithm 1: Tester.

Algorithm 1 does not always succeed. The undesired events are listed as follows:

(a) Some of x, y, z is not contained in $r_1, r_2, \ldots, r_k$. This happens with probability

$$1 - \frac{k(k-1)^2}{n(n-1)^2} = 7/8 + o(1).$$

(b) All of x, y, z are contained in $r_1, r_2, \ldots, r_k$, but $\gcd(p, q) = t$ for some $t > 1$ where $y = x + pd$ and $z = x + qd$. Note that $\hat{d} = d$ if $t = 1$. For a fixed t, this happends with probability at most

$$\frac{k(k-1)^2}{n(n-1)^2} \frac{1}{t^2} = (1/8 - o(1))/t^2.$$

By the Union bound, we have an upper bound on the probability of this undesired event.

$$(1/8 - o(1)) \sum_{t=2}^{\infty} \frac{1}{t^2} < 0.0807$$

Hence, the success rate is at least $1 - 7/8 - 0.0807 - o(1) = \delta > 0$ for some constant $\delta > 0$. If one repeats Algorithm 1 γ times so that

$$(1 - \delta)^\gamma < 1 - 0.99,$$

then the success rate is amplified to at least 0.99, as required. Hence, setting $\gamma = O(1)$ suffices. Since each execution of Algorithm 1 needs $O(n)$ time, the total running time is bounded by $O(\gamma n) = O(n)$.

6. For $k = 1$, clearly the answer is "Yes." If $k = 2$, then it suffices to check whether some pair of disks intersect. A simple algorithm can check this in $O(n^2)$ time. Otherwise $k \geq 3$, then such a k-clique, if exists, must have the centers of the corresponding k disks contained in T_{ij} where T_{ij} is defined as the intersection of two disks whose centers are those of d_i and d_j (resp.), and whose radii have length equal the distance r_{ij} between the centers of d_j and d_j, for some $i \neq j \in [1, n]$. Note that one has to ignore those T_{ij} whose $r_{ij} > 2$. For each (i, j), the disks whose centers are contained in T_{ij} can be found in $O(n)$ time. If there are at least $2k - 4$ centers in the intersection, then a k-clique exists (by the pigeon-hole principle). Otherwise, it suffices to find a largest clique in the subgraph S induced by the disks in the intersection. Note that, in this case, S contains at most $2k - 1 = O(\log n)$ nodes. As mentioned in class, subgraph S is co-bipartite. Finding a largest clique in S can be reduced to finding a maximum matching in the complement graph $\bar{S}$. By Ford-Fulkerson algorithm, this can be done in $O(k^3) = O(\log^3 n)$ time. Therefore, the total running time is

$$O(n^2(n + \log^3 n)) = O(n^3).$$