Priority Quque (continued)

We finally give the pseudo-code for implementing the binary heap. We not only maintain the array S but also the number of elements in S, denoted as n.

```
function empty($PQ$)
    if $n = 0$: return true;
    else: return true;
end function;

function insert($PQ$, $x$)
    $n = n + 1$;
    $S[n] = x$;
    bubble-up ($S$, $n$);
end function;

function find-min($PQ$)
    return $S[1]$;
end function;

function delete-min($PQ$)
    $S[1] = S[n]$;
    $n = n - 1$;
    sift-down ($S$, 1);
end function;

function decrease-key($PQ$, $k$, new-key)
    $S[k].key =$ new-key;
    bubble-up ($S$, $k$);
end function;
```

Both bubble-up and sift-down procedures runs in $O(\log n)$ time. This is because, a complete binary tree with n vertices has a height of $\log n$, while the worst case of either procedure traverses along a path between the root and a leaf. Formally, in each recursive call, k is either halved or doubled, and hence the number of recursive calls is $\log n$. The empty and find-min procedures takes $\Theta(1)$ time; the other 3 procedures takes $O(\log n)$ time, as they are dominated by either bubble-up or sift-down.

Single-Source Shortest Path Problem with Unit Edge Length

In many applications, edges of graphs are associated with *lengths*. We will consider three cases.

1. unit edge length: the length of each edge is 1;
2. positive edge length: the length of each edge is a positive number;
3. length of edge might be a negative number.

Let $G = (V,E)$ be a graph; let $l(e)$ be the length of $e \in E$ (in any of above cases). Let p be a path of G. We define the length of path p, still denoted as $l(p)$, as the sum of the length of all edges in p, i.e., $l(p) := \sum_{e \in p} l(e)$. Given $u, v \in V$, the *shortest path* from u to v, is the path from u to v with smallest length. We use notation $distance(u,v)$ to denote the length of the shortest path from u to v.

There are different variants of shortest path problems. One category is *single-source shortest path problem*, where we are given graph $G = (V,E)$ with edge length (one of the three cases) and a *source* vertex $s \in V$, and we seek to find the shortest path from s to *all* vertices.

We first solve the easiest version of shortest path problem: given $G = (V,E)$ with unit edge length and a *source* vertex $s \in V$, to find the shortest path from s to all vertices, i.e., to find $distance(s,v)$ for any $v \in V$.

The breadth-first search (BFS) can solve above problem. The idea of BFS is to traverse the vertices of graph in increasing order of their distance from s. Formally, we define V_k, $k = 0, 1, 2, \cdots$, as the subset of vertices whose distance (from s) is exactly k, i.e., $V_k = \{v \in V \mid distance(s,v) = k\}$. BFS will first traverse vertices in V_0, then vertices in V_1, then V_2, and so on, until all vertices reachable from s are traversed.

Notice that $V_0 = \{s\}$ as $distance(s,s) = 0$. How about V_1? They are the vertices reachable from s with one edge (but cannot be s), i.e., $V_1 = \{v \in V \mid (s,v) \in E\} \setminus V_0$. How about V_2? They are the vertices reachable from some vertex in V_1 with one edge, but not in V_0 or V_1, i.e., $V_2 = \{v \in V \mid u \in V_1 \text{ and } (u,v) \in E\} \setminus (V_0 \cup V_1)$. The general form is that $V_{k+1} = \{v \in V \mid u \in V_k \text{ and } (u,v) \in E\} \setminus (V_0 \cup V_1 \cup \cdots \cup V_k)$. This suggests an iterative framework of BFS: to find V_{k+1}, explore the out-edges of V_k to collect the *newly* reached vertices (i.e., those not in $V_0 \cup V_1 \cup \cdots \cup V_k$). To realize this idea, BFS uses two data structures; the complete algorithm follows.

1. Array *dist* of size $|V|$, initialized as $dist[v] = \infty$ for any $v \in V$. Array *dist* serves as two purposes: indicating a vertex has been reached or not, and storing the distance from s (after it has been reached). Formally, before v is reached, $dist[v] = \infty$; after v is reached, $dist[v] = distance\ (s,v)$.

2. Queue Q, which stores the vertices haven't been explored. Vertices will be added to Q in the order of $V_0, V_1, V_2, \cdots$, and vertices will be deleted from Q in the same order (as queue is first-in-first-out).

```
Algorithm BFS ($G = (V,E), s \in V$)
    $dist[v] = \infty$, for any $v \in V$;
    init an empty queue $Q$;
    $dist[s] = 0$;
    insert ($Q, s$);
    while (empty ($Q$) = false)
        $u$ = find-earliest ($Q$);
        delete-earliest ($Q$);
        for each edge $(u,v) \in E$
            if ($dist[v] = \infty$)
                $dist[v] = dist[u] + 1$;
                insert ($Q$, $v$);
            end if;
        end for;
    end while;
end algorithm;
```

Initially BFS has V_0 (i.e., s) in Q (right before the while loop), then BFS deletes s from Q and explores s (newly reached vertices—these are V_1, will have their *dist* set as 1 and be added to Q); at the time of finishing exploring V_0, V_1 will be in Q. Next, BFS will gradually delete and explore each of the vertices in Q (i.e., V_1); in this process, vertices in V_2 will be reached, their *dist* be set as 2, and be added to Q; after all of them are deleted and explored, Q will exactly consist of V_2.

In general, BFS keeps the following invariant: for every $k = 0, 1, 2, \cdots$, there is a time at which Q contains exactly V_k, $dist[v] = distance(s, v)$ for any $v \in V_0 \cup V_1 \cup \cdots V_k$, and $dist[v] = \infty$ for all other vertices. This invariant explains the behavior of BFS while also proves its correctness: when the Q becomes empty, $dist[v] = distance(s, v)$ for any $v \in V$.

Following above invariant and the pseudo-code, we know that the time that vertex v is reached for the first time (happend when $(u, v) \in E$ is checked and $dist[v] = \infty$), is also the time that the shortest path to v is found, that the $dist[v]$ gets assigned, and that v is added to Q.

BFS runs in $O(|V| + |E|)$ time, as each vertex will be explored at most once, and each edge will be examined at most once (for directed graph) and at most twice (for undirected graph). Note that BFS (G, s) only traverses those vertices in G can be reached from s.

Single-Source Shortest Path Problem with Positive Edge Length

We now study the single-source shortest path problem with positive edge length: given graph $G = (V, E)$ with edge length $l(e) > 0$ for any $e \in E$ and source vertex $s \in V$, to seek $distance(s, v)$ for any $v \in V$. We solve this problem with the *Dijkstra's algorithm*.

Similar to BFS, the idea of Dijkstra's algorithm is also to determine and calculate the distance of each vertex in increasing order of distance. More specifically, let $(v_1^*, v_2^*, \cdots, v_n^*)$, $n = |V|$, be the order of vertices with increasing distance, i.e., $distance(s, v_k^*) \le distance(s, v_{k+1}^*)$, $1 \le k < n$. In other words, we define v_k^* as the k-th closest vertex from s. We don't know this order in advance. But Dijktra's algorithm will identify vertices in this order and calculate their distance. For the sake of writing and notations, we define $R_k = \{v_1^*, v_2^*, \cdots, v_k^*\}$, i.e., the first k vertices in above list; R_k are also the closest k vertices from s.

Clearly, $v_1^* = s$, $distance(s, v_1^*) = distance(s, s) = 0$, and $R_1 = \{s\}$. The key question is: given R_k (i.e., the closest k vertices from s) and their distances (i.e., $distance(s, v)$ for every $v \in R_k$), how to find v_{k+1}^*? Below we show that v_{k+1}^* must be within *one-edge extension* of R_k.

Claim 1. There must exist vertex $u \in R_k$ such that $(u, v_{k+1}^*) \in E$.

Proof. Suppose conversely that for every $u \in R_k$ we have $(u, v_{k+1}^k) \notin E$. Consider the shortest path from s to v_{k+1}^*. Let w be the vertex right before v_{k+1}^* in path p, i.e, (w, v_{k+1}^*) is the last edge of p. We have $distance(s, v_{k+1}^*) = distance(s, w) + l(w, v_{k+1}^*)$. As edges have positive edge length, we have $l(w, v_{k+1}^*) > 0$, and consequently $distance(s, w) < distance(s, v_{k+1}^*)$. Besides, according to the assumption, we have $w \notin R_k$. Therefore, v_{k+1}^* cannot be the $(k+1)$-th closest vertex from s, because w has a shorter distance than v_{k+1}^*. This contradicts to the definition of v_{k+1}^*. □

Note that, in above proof, we use the fact that edges have positive lengths. Hence, above claim may not be true for graphs with negative edge length. This also explains why Dijkstra's algorithm won't work for graphs with negative edge length.

The above claim shows that, the last edge (u, v) of the shortest path from s to v_{k+1}^* satisfies that $u \in R_k$ and

$v \notin R_k$. Which such edge is the correct one? We don't know, so we enumerate all such edges $(u,v) \in E$ with $u \in R_k$ and $v \notin R_k$. Suppose that (u,v) is the last edge of the shortest path from s to v^*_{k+1}, we know that $distance(s,v) = distance(s,u) + l(u,v)$. This leads to the following formula to calculate v^*_{k+1} and its predecessor in the shortest path. See an example in Figure 1.

Corollary 1. We have

$$distance(s, v^*_{k+1}) = \min_{u \in R_k, v \in V \setminus R_k, (u,v) \in E} (distance(s,u) + l(u,v)).$$

Let (u',v') be the optimal edge in the minimization, i.e.,

$$(u',v') := \arg\min_{u \in R_k, v \in V \setminus R_k, (u,v) \in E} (distance(s,u) + l(u,v)).$$

Then $v^*_{k+1} = v'$ and u' is the predecessor of v^*_{k+1} in the shortest path from s to v^*_{k+1}.

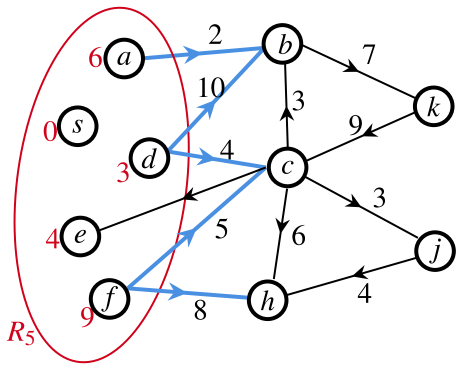


Figure 1: Example for Colollary 1. Suppose that we know $R_5 = \{s,a,d,e,f\}$ and their distances from s, marked next to vertices. To find v^*_6 and calculate $distance(s,v^*_6)$, we consider all one-edge extension of R_5, marked as thick blue edges. Following Corollary 1, $distance(s,v^*_6) = \min_{u \in R_5, v \notin R_5, (u,v) \in E}(distance(s,u) + l(u,v)) = \min\{6+2, 3+10, 3+4, 9+5, 9+8\} = 7$, and the optimal edge is (d,c). Hence, $v^*_6 = c$ and d is the predecessor of d in the shortest path.

We therefore have the following algorithm (framework), in which we follow above formula to iteratively construct R_k, $k = 1, 2, \cdots, n$. We again use array $dist$ of size n to store the distance for vertices in R_k.

```
Algorithm Dijkstra-Framework ($G = (V,E), s \in V$)
    let $R_1 = \{s\}$;
    $dist[s] = 0$; $dist[v] = \infty$ for any $v \neq s$;
    for $k = 1 \to n-1$
        calculate $(u',v') := \arg\min_{u \in R_k, v \in V \setminus R_k, (u,v) \in E}(dist[u] + l(u,v))$;
        $R_{k+1} = R_k \cup \{v'\}$;
        $dist[v'] = dist[u'] + l(u',v')$;
    end for;
end algorithm;
```

A naive implementation of above framework takes $O(|V| \cdot |E|)$ time, as for each k, the calculation of the minimization may take $O(|E|)$ time. Next lecture, we will show how to use efficient data structures to speed up, leading to the complete Dijkstra's algorithm with improved running time.

Introduction to Algorithms

Algorithm, Problem, and Instance

What's an algorithm? Algorithm is a sequence of unambiguous specifications/instructions for solving a problem. Such specifications can be followed by a human, or implemented with a machine program. The *problem* studied in theoretical computer science needs to be formally defined, usually described using input and output.

For example, consider the *sorting problem*: its input is an array $A = [a_1, a_2, \cdots, a_n]$, and its output is the sorted array, say in ascending order, of A.

A problem describes a template. It's concrete, instantiated version is called an *instance*. For example, an instance for above sorting problem would be $A = [4, 2, 10, -5, 8]$. The relationship between problem and instance is very similar to that between *class* and *object* in object-oriented programming languages. You may also think a problem is the set of all its possible instances.

Principles of Algorithm Design

There is always a straightforward algorithm to solve a problem, which is just enumerating all possible solution (and then pick the correct/best one). This is called *brute-force*. Brute-force may work for instances of size small, but in general it is *not efficient*.

We focus on designing *efficient* algorithms in this course. Here, "efficient" means "runs in polynomial-time", which we will formally define. There are two basic principles in designing efficient algorithms. Recall that an *algorithm* produces a *solution* for a *problem*.

1. *Partitioning problem into smaller subproblems.* This principle is from the perspective of problem. There are two strategies to reduce the size of a problem. First, we can partition a problem of size n into smaller subproblems with size of $n-1$; this strategy leads to *dynamic programming* algorithms. Second, we can partition a problem of size n into smaller subproblems with size of $n/2$; this strategy leads to *divide-and-conquer* algorithms.

2. *Iteratively Improving.* This principle is from the perspective of solution, i.e., we start with an initial/trivial solution, and we then gradually improve it to finally obtain the (optimal) solution. There are two strategies. First, we can start with a feasible but non-optimal solution, and then gradually improve it to achieve an optimal solutions; *linear programming* and *network flow* algorithms follow this strategy. Second, we can start with a partial-solution, and then gradually make it to a complete-solution; *greedy* algorithms follow this strategy.

Relationship between Problems

We not only design algorithms for individual problems, but also study the relationship between problems. This is very important. Studying how problems relate help in the following three folds.

1. Allow us to build the hierarchy (i.e., classes of problems) and understand them. This is the targets of *theory of computational complexity*.

2. Allow us to solve new problems using existing algorithms. For example, if we know that problem X can be transformed into problem Y (we will formally define what does this mean) and we know a good algorithm for problem Y, and we know that we can use this algorithm for Y to solve X.

3. Allow us to prove that a problem is hard to solve. For example, if we know that problem X can be transformed into problem Y, and that there does not exist any efficient algorithhm for X, then we know that there does not exist efficient algorithm for problem Y.

Example of Algorithm Design and Analysis

Let's consider the problem of *merging two sorted arrays*. The input of this problem is two sorted arrays, in ascending order, A and B, and the output is a sorted array C that consists of all elements in A and B.

An instance of this problem is: $A = [-4,2,5,8]$ and $B = [-3,2,3,4]$. For this instance, the output should be $C = [-4,-3,2,2,3,4,5,8]$.

An algorithm usually requires *data structures* to store key intermediate information. In this case, we maintain two pointers, k_A and k_B. Throughout the algorithm, we guarantee that k_A and k_B point to the smallest number in A and B that haven't been added to C.

The idea of the algorithm is to iteratively construct C (so you may call it a greedy algorithm). In each step, we compare the two numbers at the two pointers: the smaller one of the two will be the smallest one in *all* numbers that haven't been added to C; we then add it to C and update the pointers accordingly. The pseudo-code for this algorithm is given below.

```
function merge-two-sorted-arrays ($A[1\cdots m], B[1\cdots n]$)
    init an empty array $C$; (#units = 1)
    init pointers $k_A = 1$ and $k_B = 1$; (#units = 2)
    add a big number $M$ (larger than any number in $A$ and $B$) to the end of $A$ and $B$: $A[m+1] = M$ and
    $B[n+1] = M$; (think: why we do it?); (#units = 2)
    for $k = 1 \to m+n$ (#units = $m+n$)
        if $A[k_A] \leq B[k_B]$ (#units = $m+n$)
            $C[k] = A[k_A]$; (#units = $a_1$)
            $k_A = k_A + 1$; (#units = $a_2$)
        else
            $C[k] = B[k_B]$; (#units = $m+n-a_1$)
            $k_B = k_B + 1$; (#units = $m+n-a_2$)
        end if;
    end for;
    return $C$; (#units = 1)
end algorithm;
```

To analyze an algorithm, we need to (1) prove it's correct, and (2) analyze its running time.

We first prove this algorithm is correct, i.e., the resulting C is sorted and includes all numbers in A and B. This seems obvious. To give a formal and complete proof, we define the following *invariant*.

Invariant: For any $k = 0, 1, 2, \cdots, m+n$, at the end of the k-th iteration of above algorithm, we must have that C stores the smallest k numbers in A and B, k_A and k_B points to the smallest number of $A \setminus C$ and $B \setminus C$, respectively.

We now prove above invariant is correct, by *induction*. The base case is $k = 0$. At this time (i.e., before the for-loop and after the initialization), C is empty and k_A and k_B points to the first number in A and B respectively (and therefore the smallest one, as the given A and B are sorted). We now show the inductive step: assume that the invariant is correct for $k = 0, 1, \cdots, i$, we prove that the invariant is correct for $k = i+1$. The inductive assumption tells that at the end of the i-th iteration, $C[1 \cdots i]$ stores the first i smallest numbers. Consider what the algorithm does in the $(i+1)$-th iteration: it compare the numbers at the pointers, and adds the smaller one to C. As each pointer now points to the smallest one in $A \setminus C$ and $B \setminus C$, we have that the smaller one is the $(i+1)$-th smallest number in A and B. Besides, how the algorithm updates the pointers guarantees the pointers always point to the smallest one in $A \setminus C$ and $B \setminus C$, respectively. Therefore, the invariant holds after $(i+1)$-th iteration.

We then analyze its running time. The running time is measured with the *basic computing units* used by the algorithm in the *worst case*. The definition of basic computing units depends on the computing model. Usually, we think each operation like assignment, basic arithmetic operation, comparison, etc, takes 1 unit. The worst case is w.r.t. all instances, i.e., the instance takes largest number of computing units.

The required units in each step of the algorithm is marked with blue text. Let $T(m,n)$ be the running time of this algorithm when $|A| = m$ and $|B| = n$. We can write $T(m,n) = 4m + 4n + 6$.

Note that the running time of an algorithm is a function of *input size*, rather than the *input*. The input size is usually the amount of memory needed to store the actually input. In this example, array of size m can be stored in a block of memory with m memory units. An implicit assumption here is that, every number in the given arrays can be stored in a single memory unit (for example, 32 bits to store an integer in C++). This is reasonable and widely used. In certain cases, where we study very large number (which cannot be stored in 32 bits for example), then we cannot assume that such number takes 1 memory unit; instead, a (large) number x uses $\log_2 x$ bits in memory.

The running time of $T(m,n) = 4m + 4n + 6$ a brief, coarse estimation, as it depends on the computing model. Therefore, it doesn't make much sense to keep the coefficients 4 and the constants 5. Besides, we care how the running time grows as m and n grows. Hence, it suffices to write $T(m,n) = \Theta(m+n)$, meaning it grows as fast as function $m+n$. We will introduce asymptotic notations to facilitate this way of analyzing running time in next lecture.

Asymptotic Notations

Definitions and Properties

Definition 1 (Big-O)**.** Let $f = f(n)$ and $g = g(n)$ be two positive functions over integers n. We way $f = O(g)$, if there exists positive number $c > 0$ and integer $N \geq 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.

Similarly, we can define Big-O for multiple-variable functions.

Definition 2 (Big-O)**.** Let $f = f(m,n)$ and $g = g(m,n)$ be two positive functions over integers m and n. We way $f = O(g)$, if there exists positive number $c > 0$ and integers $M \geq 0$ and $N \geq 0$ such that $f(m,n) \leq c \cdot g(m,n)$ for all $m \geq M$ and $n \geq N$.

Intuitively, Big-O is analogous to "$\leq$".

Example. Let $f(m,n) = 4m+4n+5$ and $g(m,n) = m+n$. We now show that $f = O(g)$, using above definition. To show it, we need to find c, M, and N. What are good choices for them? There are lots of choices; one set of it is: $c = 7$, $M = 1$, and $N = 1$. Let's verify: $f(m,n) - c \cdot g(m,n) = 4m+4n+5-7m-7n = 5-3m-3n \leq 5-3-3 = -1 \leq 0$, where we use that $m \geq M = 1$ and $n \geq N = 1$. This proves that $f = O(g)$.

Definition 3 (Big-Omega)**.** Let $f = f(n)$ and $g = g(n)$ be two positive functions over integers n. We way $f = \Omega(g)$, if there exists positive number $c > 0$ and integer $N \geq 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq N$.

Similarly, we can define Big-Omega for multiple-variable functions.

Definition 4 (Big-O)**.** Let $f = f(m,n)$ and $g = g(m,n)$ be two positive functions over integers m and n. We way $f = \Omega(g)$, if there exists positive number $c > 0$ and integers $M \geq 0$ and $N \geq 0$ such that $f(m,n) \geq c \cdot g(m,n)$ for all $m \geq M$ and $n \geq N$.

Intuitively, Big-Omega is analogous to "$\geq$".

Example. Let $f(m,n) = 4m+4n+5$ and $g(m,n) = m+n$. We now show that $f = \Omega(g)$, using above definition. To show it, we need to find c, M, and N. We can choose: $c = 1$, $M = 0$, and $N = 0$. Let's verify: $f(m,n) - c \cdot g(m,n) = 4m+4n+5-m-n = 5+3m+3n \geq 5 \geq 0$, where we use that $m \geq M = 0$ and $n \geq N = 0$. This proves that $f = \Omega(g)$.

Claim 1. $f = O(g)$ if and only if $g = \Omega(f)$.

Proof. We have

$$\begin{aligned}
& f = O(g) \\
\Leftrightarrow\ & \exists\, c > 0, N \geq 0, \text{ s.t. } f(n) \leq c \cdot g(n), \forall n \geq N \\
\Leftrightarrow\ & \exists\, c > 0, N \geq 0, \text{ s.t. } 1/c \cdot f(n) \leq g(n), \forall n \geq N \\
\Leftrightarrow\ & \exists\, c' = 1/c > 0, N \geq 0, \text{ s.t. } g(n) \geq c' \cdot f(n), \forall n \geq N \\
\Leftrightarrow\ & g = \Omega(f)
\end{aligned}$$

□

Definition 5 (Big-Theta)**.** We say $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$.

Intuitively, Big-Theta is analogous to "$=$".

Example. Let $f(m,n) = 4m+4n+5$ and $g(m,n) = m+n$. We have $f = \Theta(g)$ as we proved that $f = O(g)$

and that $f = \Omega(g)$.

Below we give an equivalent description of Big-Theta.

Fact 1. Let f and g be two positive functions. Then $f = \Theta(g)$ if and only if $\lim_{n\to\infty} f(n)/g(n) = c > 0$.

Example. Let $f(m,n) = 4m+4n+5$ and $g(m,n) = m+n$. We now show $f = \Theta(g)$ using above fact. $\lim_{m\to\infty,n\to\infty} f/g = \lim_{m\to\infty,n\to\infty}(4m+4n+5)/(m+n) = 4 > 0$. Hence, $f = \Theta(g)$.

Definition 6 (small-o)**.** Let $f = f(n)$ and $g = g(n)$ be two positive functions. We say $f = o(g)$ if and only if $\lim_{n\to\infty} f(n)/g(n) = 0$.

Intuitively, small-o is analogous to "$<$".

Example. Let $f(n) = n$ and $g(n) = n^2$. We now show $f = o(g)$ using above definition. $\lim_{n\to\infty} f/g = \lim_{n\to\infty} n/n^2 = \lim_{n\to\infty} 1/n = 0$. Hence, $f = o(g)$.

Commonly-Used Functions in Algorithm Analysis

In theoretical computer science, we often see following categories of functions.

1. logarithmic functions: $\log\log n$, $\log n$, $(\log_n)^2$;
2. polynomial functions: $\sqrt{n} = n^{0.5}$, n, $n\log n$, $n^{1.001}$;
3. exponential functions: 2^n, $n2^n$, 3^n;
4. factorial functions: $n!$;

In above lists, any logarithmic function is small-o of any polynomial function: for example, $(\log n)^2 = o(n^{0.01})$; any polynomial function is small-o of any exponential function: for example, $n^2 = o(2^n)$; any exponential function is small-o of any factorial function: for example, $n2^n = o(n!)$. Within each category, a function to the left is small-o of a function to the right, for example $n\log n = o(n^{1.001})$.

Merge-Sort

We now start introducing the first algorithm-design technique: divide-and-conquer. A typical divide-and-conquer algorithm follows the framework below.

1. partition the original problem into smaller problems;
2. recursively solve all subproblems;
3. combine the solutions of the subproblems to obtain the solution of the original problem.

We use sorting as the first problem to demonstrate designing divide-and-conquer algorithms. Recall that the *sorting* problem is to find the sorted array (say, in increasing order) S' of a given array S. We now design a divide-and-conquer algorithm for it. For any recursive algorithm, we always need to clearly define the recursion. In this case, we define function merge-sort (S) returns the sorted array (in ascending order) of S.

The idea is to sort the first half and second half of S, by recursively call the merge-sort function. How to obtain the sorted array of S then with the two sorted half-sized arrays? We have introduced such an algorithm to merge two sorted arrays into a single sorted array. That's exactly the algorithm we need here in the combining step.

```
Algorithm merge-sort ($S[1 \cdots n]$)
    if $n \leq 1$: return $S$;
    $S_1' =$ merge-sort ($S[1 \cdots n/2]$);
    $S_2' =$ merge-sort ($S[n/2+1 \cdots n]$);
    return merge-two-sorted-arrays ($S_1', S_2'$);
end algorithm;
```

Analysis of Merge-Sort

The correctness of the algorithm can be proved by induction, as the natural structure of above algorithm is recursive. Specifically, we want to prove the statement that the merge-sort function with A as input returns the sorted array of A if $|A| = n$. The induction is w.r.t. n. The base case, i.e., $n = 1$, is clearly correct. In the inductive step, we assume that above statement is true for $|A| = 1, 2, \cdots, n-1$, and we aim to prove it is correct for $|A| = n$. As the algorithm is correct for $|A| = 1, 2, \cdots, n-1$, in particular, it is correct for $|A| = n/2$, i.e., S'_1 and S'_2 store the sorted array of S_1 and S_2, respectively. Combinining the correctness of merge-two-sorted-arrays that we have already proved, we have that merge-sort returns the sorted array of S.

To see its runing time, we define $T(n)$ as the running time of merge-sort (S) when $|S| = n$. Clearly, both merge-sort ($S[1 \cdots n/2]$) and merge-sort ($S[n/2+1 \cdots n]$) take $T(n/2)$ time. As merge-two-sorted-arrays takes linear time, we have the recurrence $T(n) = 2T(n/2) + \Theta(n)$. We will use *master's theorem* to get the closed form for $T(n)$, described below.

Master's Theorem

Master's theorem gives closed form for the following recurrence:

$$T(n) = \begin{cases} aT(n/b) + \Theta(n^d) & \text{if } n \geq 2 \\ 1 & \text{if } n \leq 1 \end{cases}$$

Master's theorem is widely used to analyze the running time of divide-and-conquer algorithms. Recall that a divide-and-conquer algorithm first partitions the original problems into subproblems, solve all subproblems recursively, and then combine them to answer the original question. Hence, above recurrence precisely describes the running time of such algorithms. Specifically, a refers to the number of subproblems that the original problem is partitioned, n/b refer to the input-size of each subproblem, and $\Theta(n^d)$ refer to the running time of the combining step. In merge-sort, $a = 2$, as it calls merge-sort twice in the algorithm, $b = 2$, as the input-size of each subproblem becomes $n/2$, and $d = 1$, as the merge-two-sorted-arrays takes $\Theta(n)$ time. Note that, it is not always the case that $a = b$ (in merge-sort though, $a = b$). Such example includes matrix multiplication.

We now solve above recurrence. Without loss of generality, we assume that n is a power of b, i.e., $n = b^k$ for some k. To further simplify, we use n^d instead of $\Theta(n^d)$. We therefore need to add $\Theta(\cdot)$ to the resulting formular of $T(n)$.

$$\begin{aligned} T(n) &= aT(n/b) + n^d \\ &= a(aT(n/b^2) + (n/b)^d) + n^d \\ &= a^2T(n/b^2) + a(n/b)^d + n^d \\ &= a^2(aT(n/b^3) + (n/b^2)^d) + a(n/b)^d + n^d \\ &= a^3T(n/b^3) + a^2(n/b^2)^d + a(n/b)^d + n^d \\ &= \cdots \\ &= a^kT(n/b^k) + \sum_{i=0}^{k-1} a^i (n/b^i)^d \end{aligned}$$

As we assume that $n = b^k$ and $T(1) = 1$, we have

$$T(n) = a^k + \sum_{i=0}^{k-1} a^i (n/b^i)^d = \sum_{i=0}^{k} a^i (n/b^i)^d = n^d \sum_{i=0}^{k} (a/b^d)^i.$$

Consider the following 3 cases.

1. If $a = b^d$, i.e., $d = \log_b a$, then $T(n) = n^d k = n^d \log_b n = \Theta(n \log n)$.
2. If $a < b^d$, i.e., $d > \log_b a$, then the series decreases exponentially, and therefore the item of $i = 0$ dominates. $T(n) = n^d a/b^d = \Theta(n^d)$.
3. If $a > b^d$, i.e., $d < \log_b a$, then the series increases exponentially, and therefore the item of $i = k$ dominates. $T(n) = n^d (a/b^d)^k = n^d a^k / b^{dk} = n^d a^k / n^d = a^k = a^{\log_b n} = n^{\log_b a}$.

We have used one facts about logarithmic function above: $a^{\log_b n} = n^{\log_b a}$.

Master's theorem can be summarized as below. For recurrence $T(n) = aT(n/b) + n^d$ with $T(1) = 1$, we have the following:

$$T(n) = \begin{cases} \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

In the case of merge-sort, $a = b = 2$ and $d = 1$. So $d = \log_b a = 1$. Hence, $T(n) = \Theta(n \log n)$.

A more generalized form of master's theorem is to solve this recurrence: $T(n) = aT(n/b) + n^d \log^s n$ with $T(1) = 1$. The closed form is given below:

$$T(n) = \begin{cases} \Theta(n^d \log^{s+1} n) & \text{if } d = \log_b a \\ \Theta(n^d \log^s n) & \text{if } d > \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Selection Problem

The *selection* problem is formally defined as follows: given an array $A = [a_1, a_2, \cdots, a_n]$ and an integer k, $1 \le k \le n$, to find the k-th smallest number in A. Here we assume that all numbers in A are distinct. The following algorithm we design can be easily extended to allow duplicated numbers in A.

A straightforward algorithm is sorting A; then the k-th element of the sorted array is exactly the k-th smallest number of A. This algorithm runs in $\Theta(n \log n)$ time. Can we do better? Notice that, when $k = 1$, this problem is to seek the smallest number in A; when $k = n$, this problem is to seek the largest number in A. In either case, we know that it can be done in linear time. In fact, the general case can also be solved in linear time, using a divide-and-conquer approach.

Pivot-based Divide-and-Conquer Algorithm

We will design a *pivot-based* divide-and-conquer algorithm. The idea is to first choose a number in A, called pivot, denoted as x. We then partition A, using x, into 3 parts, (A_1, x, A_2), where A_1 (resp. A_2) stores the numbers in A that are smaller (resp. larger) than x. Specifically, we initialize two empty lists A_1 and A_2, we then examine all numbers $a_1, a_2, \cdots, a_n$: for each $1 \le i \le n$, we put a_i to A_1 if $a_i < x$, and we put a_i to A_2 if $a_i > x$.

Now, with A_1 and A_2 in hand, we can locate which part contains the k-th smallest number of A (and therefore discard the other two parts). Specifically, if $k \le |A_1|$, then we know that the k-th smallest number of A must be in A_1, and it is exactly the k-th smallest number of A_1; if $k = |A_1| + 1$, then we know that the k-th smallest number of A must be x; if $k > |A_1| + 1$, then we know that the k-th smallest number of A must be in A_2, and it is exactly the $(k - |A_1| - 1)$-th smallest number of A_2.

Example. Let $A = [15, 5, 2, 20, 1, 9, 4, 13, 8]$ and $k = 7$. Suppose that we are given pivot $x = 8$. We partion A into A_1, x, A_2, where $A_1 = [5, 2, 1, 4]$ and $A_2 = [15, 20, 9, 13]$. As $k = 7 > |A_1| + 1 = 5$, the 7th smallest number of A must be the 2nd (i.e, $7 - 5$) smallest number of A_2.

The above analyis is illustrated with the following pseudo-code.

```
Algorithm selection ($A$, $k$)
    $x$ = find-pivot ($A$);
    partition $A$ into $A_1, x, A_2$;
    if $k \le |A_1|$: return selection ($A_1$, $k$);
    else if $k = |A_1| + 1$: return $x$;
    else: return selection ($A_2$, $k - |A_1| - 1$);
end algorithm;
```

We don't know how to find a (good) pivot yet. But no matter how we do it, the algorithm is always correct, i.e., it always find the k-th smallest number of A. The choice of x only affects the running time of this algorithm, as it affects $|A_1|$ and $|A_2|$.

Let's first have a look at the running time of above algorithm to find a clue what pivot we need to target. Let $T(n)$ be the running time of selection (A, k) when $|A| = n$. We note that this definition is independent of k; in other words, it is the running time for the *worst* choice of k. We use $P(n)$ to denote the running time of find-pivot (A, k) with $|A| = n$. Partitioning A into A_1, x, A_2 clearly takes $\Theta(n)$ time. For the remaining if-else-if-else part, again we assume the *worst-case* scenario, i.e., the larger array among A_1 and A_2 will always be chosen. Combined, we have this recurrence: $T(n) = P(n) + \Theta(n) + T\max\{|A_1|, |A_2|\})$.

Finding a Good Pivot

A good choice of pivot should result in A_1 and A_2 as balanced as possible, as in this case $\max\{|A_1|, |A_2|\}$ will be minimized. The best case, of course, is that x is the median of A which gives $|A_1| = |A_2|$. However, calculating the median of A is kind of as hard as solving the selection problem. Instead, we can try to pick a pivot x that is close to the median of A, using the idea called "median of medians".

Here is the procedure. We partition A into $n/5$ subarrays, each of which has a size of 5. We then calculate the median (i.e., the 3rd smallest number) of each subarray. We collect these medians with an array M. Clearly, $|M| = n/5$. We then calculate the *median* of M, which will be the pivot we will use.

There are two questions here. First, how to calculate the median of each subarray? As each subarray is of size 5, we can use any algorithm. For example, we can sort it, using time of $5 \cdot \log 5$, to get its median. Note, as there are $n/5$ subarrays, the total running time will be $n/5 \cdot 5 \cdot \log 5 = \log 5 \cdot n = \Theta(n)$. Second, how to

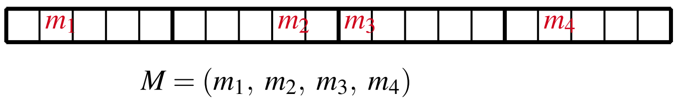


Figure 1: Finding pivot x using median of medians. The median of each subarray (of size 5) is marked with m_i. We then collect these medians and denote it as M. The median of M will be the pivot x.

calcuate the median of M? The answer is a recursive call: selection (M, $|M|/2$). We will see later on that, the resulting algorithm still runs in linear time.

The pseudo-code for find-pivot function is given below.

```
function find-pivot ($A$)
    if $|A| < 5$: find (e.g., by sorting $A$) and return the median of $A$;
    partition $A$ into $n/5$ subarrays of size 5;
    calculate the median of each subarray;
    let $M$ be the array that includes all medians;
    return selection ($M$, $|M|/2$);
end algorithm;
```

Running Time of the Selection Algorithm

By definition, the find-pivot functions takes time $\Theta(n) + T(|M|)$. Therefore, the total running time of the selection problem, in the form of a recurrence, is $T(n) = \Theta(n) + T(|M|) + \max\{T(|A_1|), T(|A_2|)\}$.

We now bound the size of $|M|$ and $\max\{|A_1|, |A_2|\}$. Clearly, $|M| = n/5$, as the number of subarrays is $n/5$. Think: how many numbers in A are *guaranteed* smaller than x (this number then gives a lower bound of $|A_1|$)? First, in these $n/5$ medians, half of them, i.e., $n/10$ numbers, are smaller than x, as x is the median of these medians. Second, consider these $n/10$ subarrays whose median is smaller than x: clearly in each of these subarrays, at least two numbers are smaller than x. This is because the median of this subarray is smaller than x and there are two numbers in this subarray that are smaller than its median. Combined, we have a lower bound of $|A_1|$: $|A_1| \geq n/10 + 2 \cdot n/10 = 3n/10$. This gives an upper bound of $|A_2|$: $|A_2| = n - |A_1| \leq 7n/10$.

Symmetrically, we have that $|A_2| \geq 3n/10$ and hence $|A_1| = n - |A_2| \leq 7n/10$. This is because, in these $n/5$ medians, $n/10$ of them are larger than x, and in these corresponding $n/10$ subarrays whose median is larger than x, there are in total $2 \cdot n/10$ numbers larger than x.

Combined, we have $\max\{|A_1|, |A_2|\} \leq 7n/10$. The above recurrence becomes $T(n) \leq \Theta(n) + T(n/5) + T(7n/10)$. How to solve this recurrence? Here is the conclusion (you will see its prove via assignment). For a more generalized version is this recurrence: $T(n) = \Theta(n) + T(c_1 n) + T(c_2 n)$, where $0 < c_1, c_2 < 1$, we have $T(n) = \Theta(n)$ if $c_1 + c_2 < 1$; $T(n) = \Theta(n \log n)$ if $c_1 + c_2 = 1$. For the selection problem we have $c_1 + c_2 = 1/5 + 7/10 < 1$. Hence its running time $T(n) = \Theta(n)$.

Choices of the Size of Subarrays

How about we partition A into subarrays of size 3? Note, in this case the algorithm is still correct. But will the algorithm still run in linear time? Let's analyze it. Now we have $|M| = n/3$, as the number of subarrays is $n/3$. In these $n/3$ medians, half of them, i.e., $n/6$ numbers, are smaller than x, and in these corresponding $n/6$ subarrays whose median is smaller than x, there are in total $1 \cdot n/6$ numbers smaller than x. This gives that $|A_1| \geq n/6 + n/6 = n/3$, which gives $|A_2| \leq 2n/3$. Symmetrically we can prove $|A_1| \leq 2n/3$ and combined we have $\max\{|A_1|, |A_2|\} \leq 2n/3$. The recursion in this case, will be $T(n) = \Theta(n) + T(n/3) + T(2n/3)$. In fact, now $T(n) = \Theta(n \log n)$ as $1/3 + 2/3 = 1$. In sum, choosing subarrays of size 3 won't give a linear time algorithm. (Note: by using the idea of "median-of-median-of-medians", a linear-time algorithm can still be obtained in this case; see assignment.)

How about we partition A into subarrays of size 7? Now we have $|M| = n/7$, as the number of subarrays is $n/7$. In these $n/7$ medians, half of them, i.e., $n/14$ numbers, are smaller than x, and in these corresponding $n/7$ subarrays whose median is smaller than x, there are in total $3 \cdot n/14$ numbers smaller than x. This gives that $|A_1| \geq n/14 + 3n/14 = 2n/7$, which gives $|A_2| \leq 5n/7$. Symmetrically we can prove $|A_1| \leq 5n/7$ and combined we have $\max\{|A_1|, |A_2|\} \leq 5n/7$. The recursion in this case, will be $T(n) = \Theta(n) + T(n/7) + T(5n/7)$. So $T(n) = \Theta(n)$ as $1/7 + 5/7 < 1$. In fact, any odd size that is larger than 5 will lead to a linear-time algorithm. But bigger size will result in bigger factor in sorting these subarrays. For example, compare size of 7 and size of 5: it takes $n/7 \cdot 7 \cdot \log 7 = \log 7 \cdot n$ time to sort in the case of size 7, which is larger than $\log 5 \cdot n$ in the case of size 5.

Randomized Algorithm for Selection Problem

We now design a *randomized algorithm* for selection problem. The idea is simply pick the pivot uniformly at random from A. The pseudo-code is given below.

```
function find-pivot-random ($A$)
    pick pivot $x$ uniformly at random from $A$;
end function ;
```

First, note that the selection algorithm combined with above random function to pick pivot is correct, i.e., it will still find the k-th smallest number of A. We now analyze its running time. Again, let $T(n)$ be the running time of selection (A, k), with above random function to select pivot, when $|A| = n$. Define random variable $Z := \max\{|A_1|, |A_2|\}$. Hence we can write $T(n) = \Theta(n) + T(Z)$. Again, here Z is a random variable, $T(Z)$ is a random variable, and therefore $T(n)$ is also a random variable.

We aim to calculate the expected running time, a common practice in analyzing randomized algorithms. We first estimate the distribution of Z. Think: what's the probability for event $Z \le 3n/4$? Answer: at least $1/2$. Why? This is because we pick x uniformly at random from x. Therefore, the probability of event of $\{x$ is between 25-percentile and 75-percentile of $A\}$ is $1/2$. And this event is equivalent to the event that $Z \le 3n/4$, according to the definition of Z. Hence, $\Pr(Z \le 3n/4) = 1/2$.

We now calculate its expected running time. We start with recursion $T(n) = \Theta(n) + T(Z)$. We first take expectation over Z on both sides: $\mathcal{E}_Z[T(n)] = \Theta(n) + \mathcal{E}_Z[T(Z)]$. Note that $T(n)$ does not contain Z (although $T(n)$ is a random variable), we have $\mathcal{E}_Z[T(n)] = T(n)$. That is $T(n) = \Theta(n) + \mathcal{E}_Z[T(Z)]$.

We now estimate $\mathcal{E}_Z[T(Z)]$.

$$\begin{aligned}
\mathcal{E}_Z[T(Z)] &= \sum_{k=n/2}^{n} \Pr(Z=k)\cdot T(k) \\
&= \sum_{k=n/2}^{3n/4} \Pr(Z=k)\cdot T(k) + \sum_{k=3n/4}^{n} \Pr(Z=k)\cdot T(k) \\
&\le T(3n/4)\cdot \sum_{k=n/2}^{3n/4} \Pr(Z=k) + T(n)\cdot \sum_{k=3n/4}^{n} \Pr(Z=k) \\
&= T(3n/4)\cdot \Pr(Z \le 3n/4) + T(n)\cdot \Pr(Z \ge 3n/4) \\
&\le T(3n/4)\cdot 1/2 + T(n)\cdot 1/2.
\end{aligned}$$

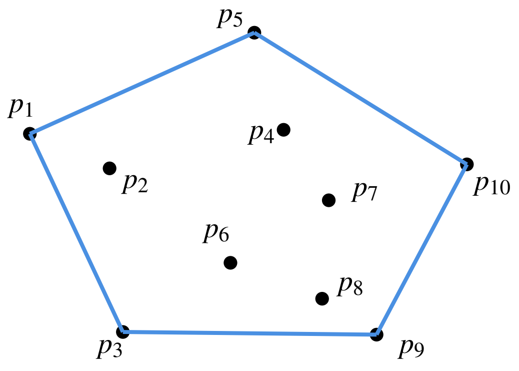
Hence, now we have $T(n) \le \Theta(n) + T(3n/4)/2 + T(n)/2$, which gives $T(n) \le \Theta(n) + T(3n/4)$. We now take expectation, over $T(n)$, on both sides: $\mathcal{E}_T[T(n)] = \Theta(n) + \mathcal{E}_T[T(3n/4)]$. By using master's theorem, we have that $\mathcal{E}_T[T(n)] = \Theta(n)$.

Convex Hull

Definitions

Definition 1 (Convex Hull)**.** Let $P = \{p_1, p_2, \cdots, p_n\}$ be a set of points on 2D plane, each of which is represented as $p_i = (x_i, y_i)$. The *convex hull* of P, denoted as $CH(P)$, is defined as the smallest, convex polygon that includes all points in P.

A convex hull is usually represented as the list of vertices (which is subset of P) of the polygon in counter-clockwise order. Such list is circular, and it's equivalent to shifting any number of vertices. If a point $p \in P$ is one of the vertices of the convex hull of P, we usually say that p is *on* the convex hull, and denote it as $p \in CH(P)$.



$$CH(P) = (p_1, p_3, p_9, p_{10}, p_5)$$

Figure 1: A set of points $P = \{p_1, p_2, \cdots, p_{10}\}$ and its convex hull.

For most computational geometry problems, we always assume that the data (points, lines, etc) given to us are in *general* situation. For example, we usually assume no three (or more) lines go through the same point, no three (or more) points are colinear, and no four points are on the same circle, etc.

Property 1. Let $p \in P$. Then $p \in CH(P)$ if and only if there exists a line l such that p is on l, and that all other points, i.e. $P \setminus \{p\}$, are on the same side of l.

The above property gives us a way to determine if a point is on the convex hull, which will be used in the following algorithm (e.g., Graham-Scan).

Property 2. For any set of points P and another point q, we have $CH(P \cup \{q\}) = CH(CH(P) \cup \{q\})$.

The above property gives a way to gradually contruct convex hull. Suppose we already know the convex hull of P, i.e., $CH(P)$. Now we want to calculate the convex hull of $P \cup \{q\}$. We only need to consider these points in $CH(P)$ and q: those points that are inside of the convex hull of P can be safely discarded.

Graham-Scan Algorithm

The idea of Graham-Scan is to gradually construct the convex hull. The first step of this algorithm is the identification of the lowest point in P, i.e., the point with smallest y-coordinate, denoted as p_*. The second step is to sort all others points in counter-clockwise order w.r.t. p_*. Specifically, for any point $p \in P$, the measure we use in sorting is the *angle* defined by points p, p_* and $(+\infty, y\text{-coordinate of } p_*)$. We define such angle for p_* is 0. All points are then sorted in ascending order by their angles. After sorting, for the sake of simplicity, we rename all points in P in this order, i.e., rename p_* as p_1, the point with second smallest angle as p_2, and so on. These two steps can be regarded as *preprocessing* steps of the Graham-Scan algorithm (i.e., first 3 lines of Algorithm Graham-Scan); the resulting of it is a sorted list of points P. See Figure 2.

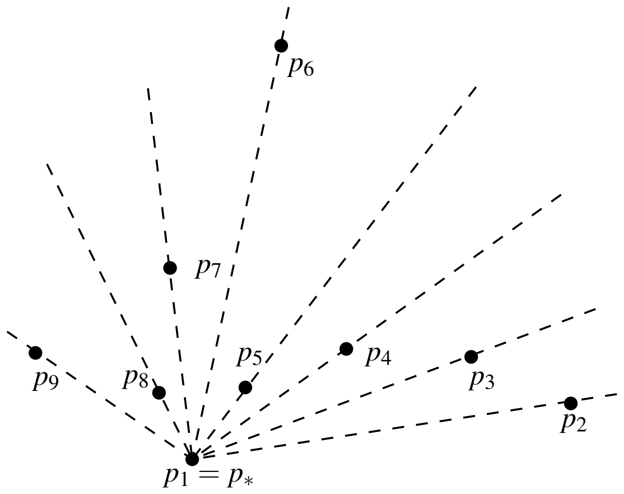


Figure 2: Preprocessing steps of Graham-Scan.

The main body of Graham-Scan is referred to as Graham-Scan-Core. It processess points in P (notice that now they are properly sorted) one by one and gradually construct the convex hull. It maintains a *stack* data structure, called S, and keeps the following variant:

Invariant: after processing the first k points of P, the convex hull of these k points, i.e., $CH(p_1, p_2, \cdots, p_k)$ will be stored in the stack S: the list of points in S from bottom to top gives the list of the vertices of the convex hull in counter-clockwise order.

How to guarantee above invariant? Consider the general case when we are processing p_k. Now we know that S stores $CH(p_1, p_2, \cdots, p_{k-1})$, and the goal is to determine $CH(p_1, p_2, \cdots, p_k)$. Following above Property 2, we know that $CH(p_1, p_2, \cdots, p_k) = CH(CH(p_1, p_2, \cdots, p_{k-1}) \cup \{p_k\})$. Hence we only need to consider the points in S and the current point p_k.

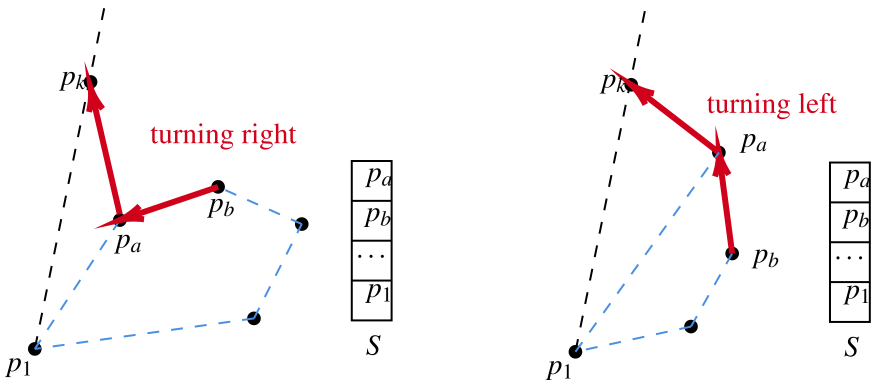


Figure 3: Procedure to decide if top element of S is on $CH(p_1, p_2, \cdots, p_k)$.

But the issue is that, points in S, which is exactly $CH(p_1, p_2, \cdots, p_{k-1}$, may not be in $CH(p_1, p_2, \cdots, p_k)$. We therefore need to *determine* and *exclude* such points in S. How? See Figure 3 for an example. We use the following procedure to determine if the top element, called p_a, of S is on $CH(p_1, p_2, \cdots, p_k)$: we check the *orientation* of the three points $p_b \to p_a \to p_k$, where p_a and p_b are the top and second top elements of S respectively. If it's "turning right", then p_a will not be on $CH(p_1, p_2, \cdots, p_k)$. Why? Because in this case p_a will be within triangle $p_1 p_b p_k$ (the angle of p_a is in the middle of that of p_b and p_k). If this happens, we can remove p_a safely, i.e., by calling *pop (S)*, as p_a cannot be in the convex hull of the first k points. After we remove p_a from S, we will need to continue to examine the top element of S iteratively. If the orientation is "turning left", then actually we get the convex hull of the first k points—that's the points in S plus p_k, where p_k will be pushed into S and become the top element of S. We will give a formal proof after the algorithm.

The complete Gramham-Scan algorithm is illustrated with pseudo-code below.

```
Algorithm Graham-Scan ($P$)
    calculate point $p_*$ in $P$ with smallest $y$-coordinate;
    sort $P$ in ascending order of the angles w.r.t. $p_*$ and $(\infty, 0)$;
    rename points in $P$ so that points in $P$ are following above order;
    Graham-Scan-Core ($P$);
end algorithm;

Algorithm Graham-Scan-Core ($P = \{p_1, p_2, \cdots, p_n\}$)
    init empty stack $S$;
    push ($S, p_1$);
    push ($S, p_2$);
    push ($S, p_3$);
    for $k = 4$ to $n$
        while ($S$ is not empty)
            let $p_a$ and $p_b$ be the top two elements of $S$;
            check the orientation of $p_b \to p_a \to p_k$;
            if "turning right": pop (S) and then continue the while loop;
            if "turning left": break the while loop;
        end while;
        push ($S, p_k$);
    end for;
end algorithm;
```

Now we show that, when the while-loop breaks, i.e., the orientation is "turning left", then $S \cup \{p_k\}$ becomes the convex hull of $\{p_1, p_2, \cdots, p_k\}$. The proof is based on the definition of convex-hull: we will verify that $S \cup \{p_k\}$ is the smallest, convex polygon that includes all $\{p_1, p_2, \cdots, p_k\}$. First, we verify that $S \cup \{p_k\}$ forms a convex polygon. This is because, all consecutive 3 points of S are "turning left", as this is guaranteed by the algorithm: we push p_k right after verifying $p_b \to p_a \to p_k$ are turning left. To verify that it is circularly "turning left", we further need to show that two more consecutive 3 points, namely, (p_a, p_k, p_1) and (p_k, p_1, p_2), are also turning left. These are easy to see as $p_k p_1$ are the rightmost line and all other points, including p_1 and p_a are on the right side of this line. Second, we verify that the convex polygon formed by $S \cup \{p_k\}$ contains all points in $\{p_1, p_2, \cdots, p_k\}$. To see this, we just need to show that all poped points are not possible to be in $CH(p_1, p_2, \cdots, p_k)$, and this is true, as every poped point is inside of a triangle (Figure 3), and therefore must be inside of the convex hull. Third, we show that $S \cup \{p_k\}$ is the smallest convex polygon that include all first k points. This is obvious as $S \cup \{p_k\}$ is a subset of the first k points.

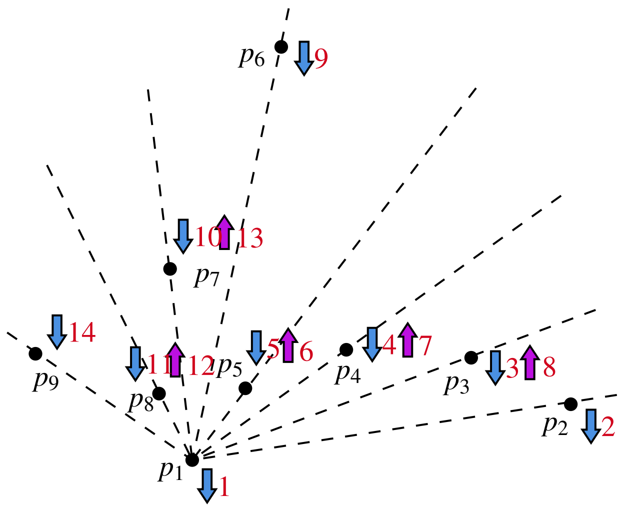


Figure 4: Running Graham-Scan-Core on the example given in Figure 2. Blue and pink arrows show the push and pop operations; the red numbers show the sequential of these operations.

The first 3 lines of Gramham-Scan algorithm corresponds to the preprocessing steps, which produce a sorted list of points. The main body of the algorithm, referred to as "Graham-Scan-Core" here, gradually construct the convex hull and guarantees above invariant. When the algorithm terminates, the convex hull of all points are stored in stack S.

Graham-Scan-Core can be regarded as an independent algorithm (you will find its use in the divide-and-conquer algorithm for convex hull). Its input is a list of points P, in which the first point, i.e. p_1, is the point with smallest y-coordinate, and the following points, i.e. $p_2, p_3, \cdots, p_n$, are placed in ascending order by their angle w.r.t. p_1 and $(+\infty, p_1.y)$. The output of Graham-Scan-Core is $CH(P)$ which will be stored in the stack S.

There are two technical details in implementing above algorithms. First, in the preprocessing step, for each point p_i do we really need to calculate the actual angle of $\angle p_i p_* p_\infty$, where $p_\infty = (+\infty, p_*.y)$? No. Notice that the only purpose of calculating these angles is sorting these points. Therefore, only the relative order matters but not the actual angles. Hence, we can calculate $\cos(\cdot)$ of the angle and use it to sort, as $\cos(\cdot)$ is monotone over $[0, \pi]$. Clearly, $\cos(\angle p_i p_* p_\infty) = \overrightarrow{p_* p_i} \cdot (1,0) / |\overrightarrow{p_* p_i}|$.

Second, in Graham-Scan-Core, how to determine the orientation of 3 points (i.e. $p_b \to p_a \to p_k$)? We will use "right-hand-rule". Define two vectors: $\overrightarrow{A} := \overrightarrow{p_b p_a}$ and $\overrightarrow{B} = \overrightarrow{p_a p_k}$. Let $\overrightarrow{A} = (x_A, y_A)$ and $\overrightarrow{B} = (x_B, y_B)$ be their coordinates, which can be calculated from the coordinates of p_b, p_a and p_k. The right-hand-rule gives that

$$\begin{cases} x_A y_B - x_B y_A > 0 & \text{if and only if} \quad p_b \to p_a \to p_k \text{ "turning left"} \\ x_A y_B - x_B y_A < 0 & \text{if and only if} \quad p_b \to p_a \to p_k \text{ "turning right"} \\ x_A y_B - x_B y_A = 0 & \text{if and only if} \quad p_b, p_a, p_k \text{ "colinear"} \end{cases}$$

Running Time of Graham-Scan

Although Graham-Scan-Core consists of two nested loop, in fact it runs in linear time! To see this, consider the types of operations and how many each type of operation the algorithm needs to do. There are three types of operations this algorithm does: *push*, *pop*, and *checking-orientation*, and each operation takes $\Theta(1)$ time. Now let's consider the times of each type will need to execute. First, notice that #(*push*) $= n$, as each point will be pushed to the stack exactly once. Second, #(*pop*) $\leq n$, as #(*push*) $\leq$ #(*pop*) starting from an empty stack (a point must be pushed into the stack prior to pop). Third, notice that right after *checking-orientation* it is either a *push* or a *pop* operation (i.e., it's not possible to have two *checking-orientation* operations next to each other). This implies that #(*checking-orientation*) $\leq$ #(*push*) + #(*pop*) $\leq 2n$. Combined, we have that Graham-Scan-Core takes $\Theta(n)$ time.

The preprocessing step of Graham-Scan takes $\Theta(n\log n)$, as it's dominated by the sorting step. The entire running time of Graham-Scan therefore takes $\Theta(n\log n)$ time.

Divide-and-Conquer Algorithm for Convex Hull

We now design a divide-and-conquer algorithm for convex hull problem. As usual, we define recursive function, say, CHDC (P), which takes a set of points P as input and returns $CH(P)$, represented as the list of vertices of the convex polygon in counter-clockwise order.

```
function CHDC ($P = \{p_1, p_2, \cdots, p_n\}$)
    if $n \leq 3$: resolve this base case and return the convex hull;
    $C_1$ = CHDC ($P[1..n/2]$);
    $C_2$ = CHDC ($P[n/2+1..n]$);
    return combine ($C_1, C_2$);
end algorithm;
```

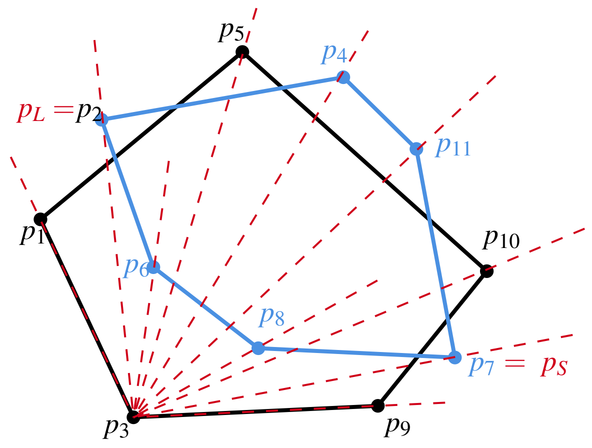
The "combine" function in above pseudo-code is missing. Before design an algorithm for "combine", we first make sure that it suffices to only consider points in C_1 and C_2. Formally, we can write $CH(P) = CH(C_1 \cup C_2)$. This can be proved using the Property 2 of convex hull. Intuitively, any point *inside* C_1 or C_2 with be inside $CH(P)$, and therefore won't be *on* $CH(P)$.

We can use, for example, Graham-Scan to calculate $CH(C_1 \cup C_2)$, which takes $\Theta(m\log m)$, where $m = |C_1 \cup C_2|$. Can we do better? Yes. We will design an $\Theta(m)$ time algorithm to calculate $CH(C_1 \cup C_2)$. The idea is to take advantage of that, C_1 and C_2 are already sorted (in counter-clockwise order along the polygon), and then to use Graham-Scan-Core to calculate $CH(C_1 \cup C_2)$. Recall that, Graham-Scan-Core is linear-time algorithm for convex hull; it just requires that all points in its input is sorted in counter-clockwise order w.r.t. the first point in its input.

The pseudo-code for combine procedure is given below.

```
function combine ($C_1, C_2$)
    find $p_*$ in $C_1 \cup C_2$ with smallest $y$-coordinate;
    assume that $p_* \in C_1$; otherwise exchange $C_1$ and $C_2$;
    rewrite $C_1$ into $C'_1$ with circular shifting so that $p_*$ is the first point of $C'_1$; (*Note: now points in $C'_1$ is sorted w.r.t. $p_*$ in counter-clockwise order.*)
    find $p_S$ in $C_2$ with smallest angle (i.e., angle $\angle p_S p_* p_\infty$, where $p_\infty = (+\infty, p_*.y)$);
    find $p_L$ in $C_2$ with largest angle (i.e., angle $\angle p_L p_* p_\infty$, where $p_\infty = (+\infty, p_*.y)$);
    let $C_{2a}$ be the sublist of $C_2$ from $p_S$ to $p_L$ in counter-clockwise order; (*Note: now points in $C_{2a}$ is sorted w.r.t. $p_*$ in counter-clockwise order.*)
    let $C_{2b}$ be the sublist of $C_2$ from $p_S$ to $p_L$ in clockwise order; (*Note: now points in $C_{2b}$ is sorted w.r.t. $p_*$ in counter-clockwise order.*)
    $C'_2$ = merge-two-sorted-arrays ($C_{2a}, C_{2b}$); (*Note: merging is by the angle w.r.t. $p_*$ and $p_\infty$; after it points in $C'_2$ is sorted w.r.t. $p_*$ in counter-clockwise order.*)
    $C'$ = merge-two-sorted-arrays ($C'_1, C'_2$); (*Note: merging is by the angle w.r.t. $p_*$ and $p_\infty$; after it points in $C'$, i.e., points in $C_1 \cup C_2$, is sorted w.r.t. $p_*$ in counter-clockwise order.*)
    return Graham-Core-Scan ($C'$);
end algorithm;
```

See figure below for explaining above combine step.



$C_1 = (p_5, p_1, p_3, p_9, p_{10})$ $C_2 = (p_4, p_2, p_6, p_8, p_7, p_{11})$

$C'_1 = (p_3, p_9, p_{10}, p_5, p_1)$ $C_{2a} = (p_7, p_{11}, p_4)$

$C_{2b} = (p_8, p_6, p_2)$

$C'_2 = (p_7, p_8, p_{11}, p_4, p_6, p_2)$

$C' = (p_3, p_9, p_7, p_{10}, p_8, p_{11}, p_4, p_5, p_6, p_2, p_1)$

Figure 1: Example of combining C_1 and C_2.

Clearly each step of above combine takes linear time. Therefore, the entire running time of combine is $\Theta(|C_1| + |C_2|)$.

To obtain the running time of CHDC, we write its recursion $T(n) = 2T(n/2) + \Theta(|C_1| + |C_2|)$. In worst case, $|C_1| + |C_2| = \Theta(n)$, and therefore $T(n) = 2T(n/2) + \Theta(n)$, which gives $T(n) = \Theta(n \log n)$. In this (worst) case the running time of this divide-and-conquer algorithm is the same with Graham-Scan. However, if the size of $|C_1| + |C_2|$ is in the order of $o(n)$, we will have improved running time. For example, if $|C_1| + |C_2| = \Theta(n^{0.99})$, then $T(n) = \Theta(n)$. This might happen in practical cases, although in worst case divide-and-conquer runs in $\Theta(n \log n)$ time.

Half-Plane Intersection

Definitions

Definition 1 (half-planes)**.** A line l on 2D plane with function $y = ax - b$ defines two *half-planes*: the *upper half-plane*: $y \geq ax - b$ and the *lower half-plane*: $y \leq ax - b$.

Definition 2 (upper- and lower-envelop)**.** Let $L = \{y = a_i x - b_i \mid 1 \leq i \leq n\}$ be a set of lines on 2D plane. We define the *upper-envelop* of L, denoted as $UE(L)$, as the intersection of the corresponding n upper half-planes $\{y \geq a_i x - b \mid 1 \leq i \leq n\}$. We define the *lower-envelop* of L, denoted as $LE(L)$, as the intersection of the corresponding n lower half-planes $\{y \leq a_i x - b \mid 1 \leq i \leq n\}$.

Either upper-envolop or lower-envelop of a set of lines can be represented as the list of lines that define its boundary from left to right. In the example below, we can write $UE(L) = (l_1, l_2, l_4, l_7)$ and $LE(L) = (l_7, l_5, l_1)$.

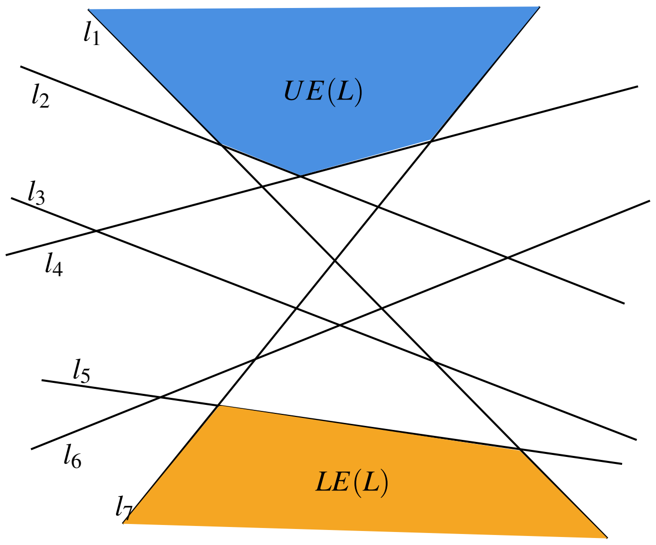


Figure 1: Illustration of upper-envelop and lower-envelop of lines $L = \{l_1, l_2, \cdots, l_7\}$.

We want to design efficient algorithms to calculate the upper- and lower-envelop of a set of lines. In fact, we don't need to design any new algorithm here. Below we will show that, the problem of finding upper- and lower-envolop of a set of lines is equivalent to the problem of finding the convex hull of a set of points. Therefore, the algorithms we've designed for finding the convex hull can be directly used to find the upper- and lower-envelop of lines.

Duality

Definition 3 (dual of a point)**.** Let $p = (p_x, p_y)$ be a point on 2D plane. We define the *dual* of p, denoted as p^*, as a line with function $y = p_x x - p_y$ on 2D plane.

Definition 4 (dual of a line)**.** Let l be a line with function $y = ax - b$ on 2D plane. We define its *dual*, denoted as l^*, as a point with coordinates (a, b) on 2D plane.

The following three properties are direct consequences of above definitions. (Think how to prove them.)

Property 1. For any point p, we have $(p^*)^* = p$. For any line l, we have $(l^*)^* = l$.

Property 2. Point p is on line l if and only if point l^* is on line p^*.

Property 3. Point p is above (resp. below) line l if and only if point l^* is above (resp. below) line p^*.

Half-plane Intersection vs. Convex Hull

Definition 5 (upper- and lower-hull)**.** Let P be a set of points, and let $CH(P)$ be the convex hull of P. Let $p_S \in CH(P)$ be the vertex with smallest x-coordinate, and $p_L \in CH(P)$ be the vertex with largest x-coordinate. Therefore p_S and p_L partition $CH(P)$ into two parts: the list of vertices from p_S to p_L following the counter-clockwise order is called *lower hull* of P, denoted as $LH(P)$; the list of vertices from p_L to p_S following the counter-clockwise order is called *upper hull* of P, denoted as $UH(P)$.

We now show that upper- and lower-envelop of lines is essentially the same with lower- and upper-hull of points. We first prove the connection between upper-envelop and lower-hull; the other one, i.e., lower-envelop and upper-hull, can be proved symmetrically.

Let L be a set of lines, we define $L^* = \{l^* \mid l \in L\}$, i.e., the set of "dual points" of L.

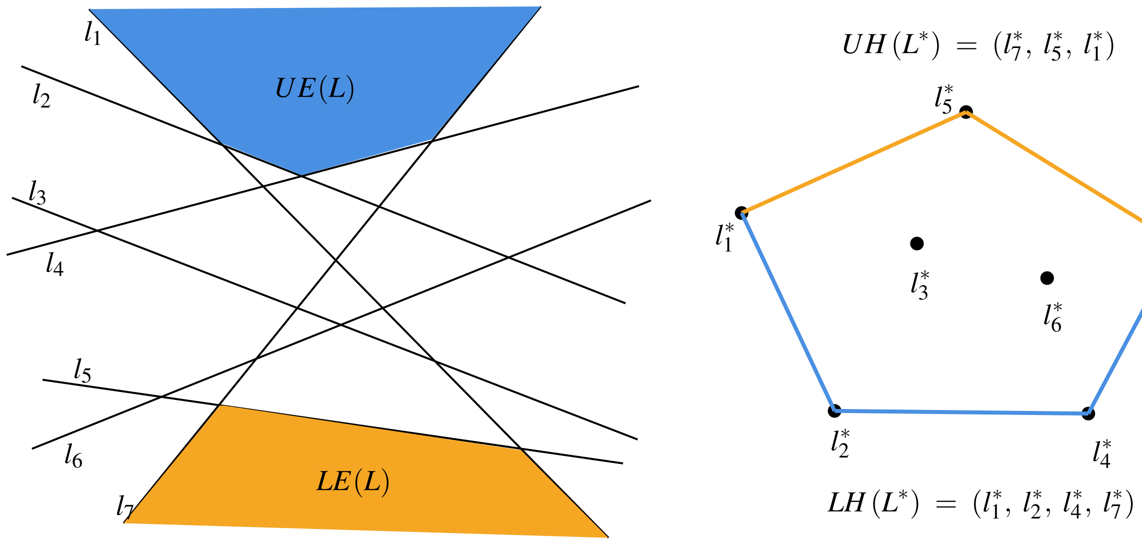


Figure 2: Illustration of duality between upper-/lower-envelop and lower-/upper-hull.

Claim 1. A line $l \in L$ is part of the boundary of $UE(L)$ if and only if l^* is one of the vertices of $LH(L^*)$.

Proof. Line l is part of $UE(L)$, implies that a piece of l is above all other lines. This is equivalent to: there exists a point p, such that p is on l, and that p is above all lines in $L \setminus \{l\}$. This statement is also equivalent to the following statement, by translating everything to their dual counterparts (and applying above Properties of duality): there exists a line p^*, such that l^* is on p^* and that all points in $L^* \setminus \{l^*\}$ are above line p^*. Clearly, this statement is also equivalent to that l^* is one vertex of the lower-hull of L^* (think the Properties of convex hull). $\square$

The above claim shows that lines in $UE(L)$ and vertices in $LH(L^*)$ are in a "dual" relationship. We now show how their ordering are connected. Recall that we represent $UE(L)$ as a list of lines from left to right. Therefore, the *slope* of these lines are in increasing order. As the dual of line $y = ax - b$ is point (a, b), i.e., the slope of a line becomes the x-coordinate of its dual, we know that the corresponing "dual points" of $UE(L)$ are in the increasing order of their x-coordinates.

The above two facts can be combined as the following: $UE(L) = (l_{p_1}, l_{p_2}, \cdots, l_{p_k})$ if and only if $LH(L^*) = (l^*_{p_1}, l^*_{p_2}, \cdots, l^*_{p_k})$. Formally, we can write

Fact 1. $UE(L) = (LH(L^*))^*$.

Symmetrically, with the same reasonging, we can prove that $LE(L) = (l_{p_1}, l_{p_2}, \cdots, l_{p_k})$ if and only if $UE(L^*) = (l^*_{p_1}, l^*_{p_2}, \cdots, l^*_{p_k})$. (Recall that $LE(L)$ is represented as the list of lines from left to right, i.e., their slopes are decreasing, while $UH(L^*)$ is represented as the list of vertices from rightmost vertex to leftmost vertex in couter-clockwise order, i.e., their x-coordinates are also decreasing.) Formally, we can also write

Fact 2. $LE(L) = (UH(L^*))^*$.

Graph: Definitions and Representations

A graph is usually denoted as $G = (V, E)$, where V represents vertices, and E represents edges. There are two types of graphs, directed ones and undirected ones (see Figures below).

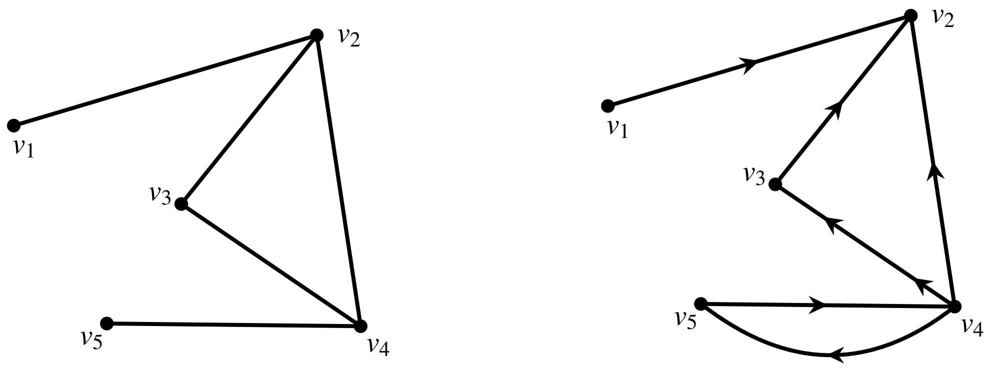
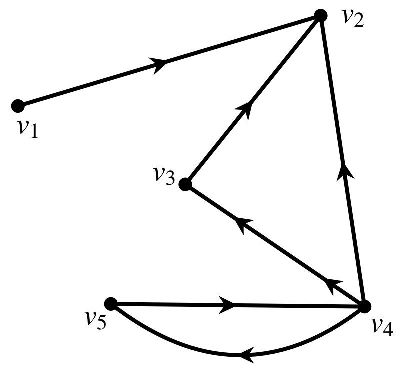


Figure 1: Left: a undirected graph $G = (V, E)$, where $V = \{v_1, v_2, v_3, v_4, v_5\}$, and $E = \{(v_1, v_2), (v_3, v_2), (v_4, v_3), (v_4, v_2), (v_4, v_5)\}$. Right: a directed graph $G = (V, E)$, where $V = \{v_1, v_2, v_3, v_4, v_5\}$, and $E = \{(v_1, v_2), (v_3, v_2), (v_4, v_3), (v_4, v_2), (v_4, v_5), (v_5, v_4)\}$.

Notice that for an edge (v_i, v_j) in an undirected graph, the order of v_i and v_j are interchangable, i.e., $(v_i, v_j) = (v_j, v_i)$. In directed graph this is not the case.

Adjacency matrix and adjacency list are two commonly-used data structures to represent a graph. Adjacency matrix uses a binary matrix M of size $|V| \times |V|$ to store a graph $G = (V, E)$: $M[i, j] = 1$ if and only if $(v_i, v_j) \in E$. This definition applies to both directed graphs and undirected graphs.

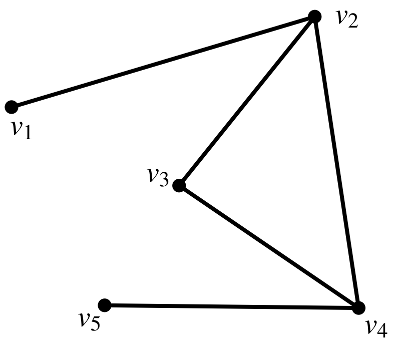
For undirected graphs, clearly the adjacency matrix M is symmetric. If we assume that there is no "self-loop" edges in the form of (v_i, v_i), then the number of "1"s in M is exactly $2|E|$ for undirected graph. The number of "1"s in M is exactly $|E|$ for directed graph.



	v_1	v_2	v_3	v_4	v_5
v_1	0	1	0	0	0
v_2	0	0	0	0	0
v_3	0	1	0	0	0
v_4	0	1	1	0	1
v_5	0	0	0	1	0

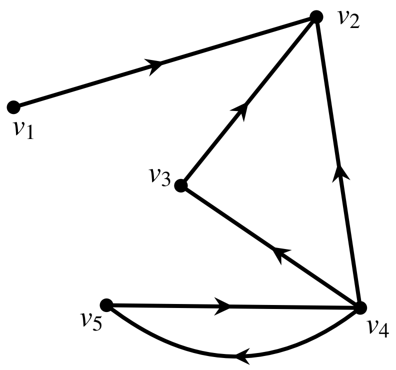
Figure 2: Adjacency matrix representation (directed graph).

Adjacency list maintains a list/array A_i for each vertex $v_i \in V$, where A_i stores $\{v_j \in V \mid (v_i, v_j) \in E\}$, i.e., the adjacent edges/vertices of v_i. A pointer is usually maintained for each vertex v_i that points to the array A_i. Clearly, for undirected graph, $\sum_{v_i \in V} |A_i| = 2|E|$, assuming that there is no "self-loop" edges. For directed graph, $\sum_{v_i \in V} |A_i| = |E|$.



$$\begin{array}{c|ccccc} & v_1 & v_2 & v_3 & v_4 & v_5 \\ \hline v_1 & 0 & 1 & 0 & 0 & 0 \\ v_2 & 1 & 0 & 1 & 1 & 0 \\ v_3 & 0 & 1 & 0 & 1 & 0 \\ v_4 & 0 & 1 & 1 & 0 & 1 \\ v_5 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Figure 3: Adjacency matrix representation (undirected graph).



$v_1 \to (v_2)$

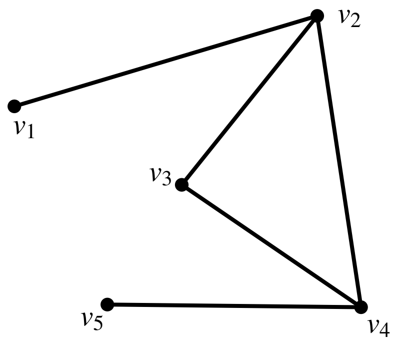
$v_2 \to ()$

$v_3 \to (v_2)$

$v_4 \to (v_2,\ v_3,\ v_5)$

$v_5 \to (v_4)$

Figure 4: Adjacency list representation (directed graph).



$v_1 \to (v_2)$

$v_2 \to (v_1,\ v_3,\ v_4)$

$v_3 \to (v_2,\ v_4)$

$v_4 \to (v_2,\ v_3,\ v_5)$

$v_5 \to (v_4)$

Figure 5: Adjacency list representation (undirected graph).

Which one is better, adjacency matrix or adjacency list? Let's consider some measures.

- The space complexity. Clearly, the adjacency matrix needs $\Theta(|V|^2)$ space to store. The adjacency list can be stored in $\Theta(|V|+|E|)$ space, where $\Theta(|V|)$ is used to store all $|V|$ pointers, and $\Theta(|E|)$ is used to store all arrays $\{A_i\}$ as we've seen $\sum_{v_i \in V} |A_i| = |E|$. Therefore, for the space complexity, adjacency list is better, as $|E| = O(|V|^2)$; in particular in *sparse* graphs, $|E|$ will be way smaller than $|V|^2$.

- Querying if $(v_i, v_j) \in E$. Given v_i and v_j, whether $(v_i, v_j) \in E$ can be done in $\Theta(1)$ time if the graph G is represented with an adjacency matrix, as this can be answered by a direct access to $M[i, j]$. If G is represented with an adjacency list, to check if $(v_i, v_j) \in E$, one needs to tranverse A_i and see if v_j

is in A_i or not, and clearly this takes $\Theta(|A_i|)$ time. Note that, if we assume that the vertices in A_i are sorted, then searching for the appearance of v_j in A_i can be done through *binary search* which takes $\Theta(\log |A_i|)$ time. In any case, adjacency matrix is better in fastly querying.

- Listing adjacent vertices of a vertex. Given v_i, one needs to tranverse the entire row (the i-th row) of the adjacency matrix to find all adjacent vertices of v_i which takes $\Theta(|V|)$ time. In case of adjacency list, one just needs to return the list A_i which takes $\Theta(|A_i|)$ time. Hence, adjacency list is better in listing adjacent vertices.

In practice, adjacency list is usually the first choice, for an obvious reason that, for huge graphs, it is not possible to store an $|V| \times |V|$ matrix in memory.

Connectivity of Graphs

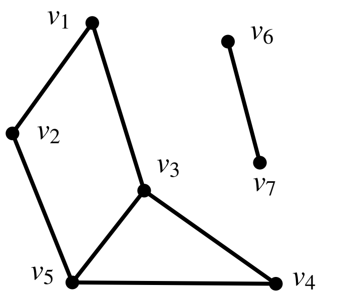
A *path* in a graph is a list of consecutive edges. In directed graphs, two consecutive edges in a path must be in the form of $(v_a, v_b), (v_b, v_c)$, i.e., the head of the predecessor connects to the tail of successor. If there exists a path from u to v, then we also say u can reach v, or v is reachable from u, or v can be reached from u. These definitions applies to both directed and undirected graphs.

One basic procedure in graphs is to find the set of vertices that are reachable from a given vertex. We will use an array, called *visited*, of size $|V|$, to store the vertices that are reachable from the given vertex v_i: $visited[j] = 1$ if and only if there exists a path from v_i to v_j. This array will be initialized as 0 for all entries. The following recursive algorithm, named *explore*, finds all vertices that are reachable from v_i and stores these vertices in *visited* array properly.

```
function explore ($G = (V,E), v_i \in V$)
    $visited[i] = 1$;
    for any edge $(v_i, v_j) \in E$
        if ($visited[j] = 0$): explore ($G, v_j$);
    end for;
end algorithm;
```

DFS and Connectivity of Graphs

Recall that the *explore* function starting from a given vertex v finds all vertices in G reachable from v. Below we give two examples of running *explore*.



input graph: $G = (V,E)$

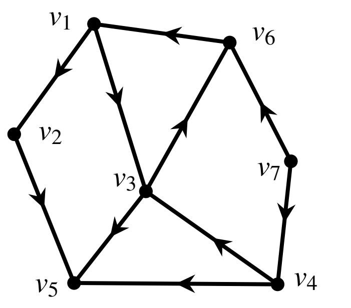
$v_1 \rightarrow (v_2,\ v_3)$
$v_2 \rightarrow (v_1,\ v_5)$
$v_3 \rightarrow (v_1,\ v_4, v_5)$
$v_4 \rightarrow (v_3,\ v_5)$
$v_5 \rightarrow (v_2,\ v_3,\ v_4)$
$v_6 \rightarrow (v_7)$
$v_7 \rightarrow (v_6)$

adjacency list of G

final array of *visited*

explore $(G,\ v_1)$	1	1	1	1	1	0	0

Figure 1: Running *explore* (G, v_1) on an undirected graph.



input graph: $G = (V,E)$

$v_1 \rightarrow (v_2, v_3)$
$v_2 \rightarrow (v_5)$
$v_3 \rightarrow (v_5,\ v_6)$
$v_4 \rightarrow (v_3,\ v_5)$
$v_5 \rightarrow ()$
$v_6 \rightarrow (v_1)$
$v_7 \rightarrow (v_4, v_6)$

adjacency list of G

final array of *visited*

explore $(G,\ v_1)$	1	1	1	0	1	1	0

Figure 2: Running *explore* (G, v_1) on an directed graph.

We now define "connected" and "connected component" to formally reveal the connectivity-structure of graphs. Let $u, v \in V$. We say u and v are *connected* if and only if there exists a path from u to v *and* there exists a path from v to u. We note that this definition applies to both directed and undirected graph. In undirected graph, the existence of a path from u to v implies the existence of a path from v to u. However, this is not necessarily true in directed graphs. For example, in Figure 2, there exists a path from v_1 to v_5 but there is no path from v_5 to v_1 (so they are not connected).

Let $G = (V,E)$. Let $V_1 \subset V$. We say V_1 is a *connected component* of G, if and only if (1), for *every pair of* $u, v \in V_1$, u and v are connected, and (2), V_1 is *maximal*, i.e., there does not exists vertex $w \in V \setminus V_1$ such that $V_1 \cup \{w\}$ satisfies condition (1). For example, in Figure 2, $\{v_1, v_3, v_6\}$ is a connected component; $\{v_2\}$

is a connected component; $\{v_1, v_3\}$ is not a connected component (as it is not maximal, i.e., does not satisfy condition 2).

The explore algorithm identifies all vertices that can be reached from v_i. Hence, in the case of undirected graphs, these vertices (including v_i) are pairwise reachable, i.e., they form a connected component of G (summarized below). In other words, $explore(G, v_i)$ identifies the connected component of G that includes v_i.

Fact 1. For undirected graphs, after $explore(G, v_i)$, the vertices that are marked by *visited*, i.e., $\{v_j \mid visited[j] = 1\}$ forms a connected component of G that includes v_i.

However, the above fact does not apply to directed graph: Figure 2 gives such an example, where $\{v_1, v_2, v_3, v_5, v_6\}$ does not form a connected component. Note: in directed graphs $\{v_j \mid visited[j] = 1\}$ are still those vertices that are reachable from v_i; it's just that they may not be a connected component of G.

How to identify *all* connected components of an undirected graph? We can run above explore algorithm multiple times, each of which starts from an un-explored vertex, until all vertices are explored. To keep track of which vertices are in which connected component, we will introduce variable *num-cc* to store the index of current connected component. We redefine the behavior of *visited* array: $visited[j] = 0$ still represents that v_j has not yet been explored; $visited[j] = k$, $k \geq 1$, represents that v_j has been explored and v_j is in the k-th connected component.

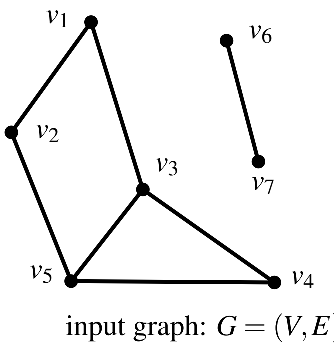
This new algorithm that traverses all vertices and edges of a graph is named as DFS (depth first search). We also slightly changed the explore function, which allows to store which connected component each vertex is in. The pseudo-codes are given below.

```
function DFS ($G = (V, E)$)
    num-cc = 0;
    $visited[i] = 0$, for all $1 \leq i \leq |V|$;
    for $i = 1 \to |V|$
        if ($visited[i] = 0$)
            num-cc = num-cc + 1;
            explore ($G, v_i$);
        end if;
    end for;
end algorithm;
```

```
function explore ($G = (V, E), v_i \in V$)
    $visited[i]$ = num-cc;
    for any edge $(v_i, v_j) \in E$
        if ($visited[j] = 0$): explore ($G, v_j$);
    end for;
end algorithm;
```

Below we gave examples of running DFS on undirected graphs and undirected graphs.

DFS runs in $\Theta(|E| + |V|)$ time. This is because, each vertex is explored exactly once, and each edge is examined exactly once (in the case of directed graphs) or exactly twice (in the case of undirected graphs).



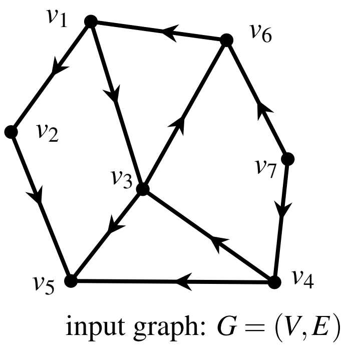
$v_1 \to (v_2,\ v_3)$
$v_2 \to (v_1,\ v_5)$
$v_3 \to (v_1,\ v_4, v_5)$
$v_4 \to (v_3,\ v_5)$
$v_5 \to (v_2,\ v_3,\ v_4)$
$v_6 \to (v_7)$
$v_7 \to (v_6)$

adjacency list of G

final array of *visited* after running $DFS(G)$

1	1	1	1	1	2	2

Figure 3: Running $DFS\ (G)$ on an undirected graph.



$v_1 \to (v_2, v_3)$
$v_2 \to (v_5)$
$v_3 \to (v_5,\ v_6)$
$v_4 \to (v_3,\ v_5)$
$v_5 \to ()$
$v_6 \to (v_1)$
$v_7 \to (v_4, v_6)$

adjacency list of G

final array of *visited* after running $DFS(G)$

1	1	1	2	1	1	3

Figure 4: Running $DFS\ (G)$ on a directed graph.

Fact 2. For undirected graphs, DFS (G) identifies all connected components of G: $\{v_j \mid visited[j] = k\}$ constitutes the k-th connected component of G.

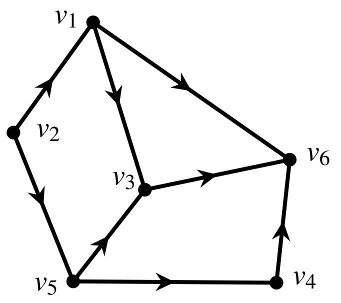
Again, the above fact does not apply to directed graphs: $\{v_j \mid visited[j] = k\}$ are not necessarily form a connected component, although you can still run DFS on directed graphs. In Figure 4, $\{v_j \mid visited[j] = 1\}$ gives such an counter-example. To prepare revealing the connectivity-structure of directed graphs, we first introduce a spsecial class of directed graphs, given below.

Directed Acyclic Graph (DAG)

Definition 1 (DAG)**.** A directed graph $G = (V,E)$ is *acyclic* if and only if G does not contain cycles.

Definition 2 (Linearization / Topological Sorting)**.** Let $G = (V,E)$ be a directed graph. Let X be an ordering of V. If X satisfies: if $(v_i, v_j) \in E$, then v_i is before v_j in X, then we say X is a linearization (or toplogical sorting) of G.

See some examples below.



$(v_2, v_1, v_5, v_3, v_4, v_6)$	linearization? Yes
$(v_2, v_5, v_1, v_4, v_3, v_6)$	linearization? Yes
$(v_2, v_5, v_3, v_4, v_1, v_6)$	linearization? No, see edge (v_1, v_3)
$(v_2, v_1, v_4, v_5, v_3, v_6)$	linearization? No, see edge (v_5, v_4)

Figure 5: Examples of linearization.

If a directed graph G admits a linearization, then we say G can be *linearized*. We now show that linearization is an *equivalent* characterization of DAGs.

Claim 1. A directed graph G can be linearized if and only if G is a DAG.

Proof. Let's first prove that if G can be linearized, then G is a DAG. This is equivalent to proving its contraposition: if G contains a cycle, then G cannot be linearized. Suppose that there exists an cycle $v_{i_1} \to v_{i_2} \to \cdots \to v_{i_k} \to v_{i_1}$ in G. Then the linearization X must satisfy that v_{i_j} is before $v_{i_{j+1}}$ for all $j = 1, 2, \cdots, k-1$, and that v_{i_k} is before v_{i_1}, in X. Clearly, this is not possible.

The other side of the statement, i.e., if G is a DAG, then G can always be linearized, can be proved constructively. We will design an algorithm (see below), that constructs a linearization for any DAG. $\square$

Directed Acyclic Graphs

Let $G = (V,E)$ be a directed graph. If a vertex $v \in V$ does not have any in-edges (i.e., *in-degree* is 0), we call it *source vertex*; if a vertex $v \in V$ does not have any out-edges (i.e., *out-degree* is 0), we call it *sink vertex*.

A directed graph may contain multiple source vertices or sink vertices, or may not have any source vertex or sink vertex. (Can you give such examples?)

Claim 1. A DAG $G = (V,E)$ always has source vertex and sink vertex.

Proof. Let's prove it by contradiction. Assume that G does not contain any source. First, G must not contain self-loop as otherwise G won't be a DAG. Let v be any vertex in V. As v is not a source, we know that there exists some vertex u points to v, i.e., $(u,v) \in E$. Now since u is not a source then there must exist another vertex w such that $(w,u) \in E$. Notice that $w \neq v$ as otherwise there will be a cycle: $v = w \to u \to v$. This means that w is a new vertex. Again as w is not a source, there must exist another *new* vertex points to it. This process can be extended infinitely following the fact and assumption that G is a DAG and all vertices not are sources, but this is not possible as the number of vertices is limited. The existence of sink can be proved symmetrically. □

Following above fact, we can design an algorithm to find a linearization of a DAG: it iteratively finds source vertex and removes it and its adjacent edges.

```
Algorithm find-linearization ($G = (V,E)$)
    init $X$ as empty list;
    while ($|X| < |V|$)
        arbitrarily find a source vertex $u$ of $G$;
        add $u$ to the end of $X$;
        remove $u$ and its adjacent edges;
    end while;
end algorithm;
```

This algorithm is correct. First, when a vertex u is added to X, it is a source vertex of the current graph, which means that $\{w \mid (w,u) \in E\}$ is either empty or all of them have been added to X. Second, X will include all vertices. This is because, a source always exists in a DAG (as we just proved). The above algorithm is more a framework, as how we update the graph is not given specifically, and which affects the running time.

Above algorithm also gives a constructive proof that, a DAG can always be linearized. This completes the proof stated in previous lecture: a directed graph is a DAG if and only if it can be linearized.

Claim 2. Let X be any linearization of a DAG. Then the first vertex of X is a source vertex and the last vertex of of X is a sink vertex.

Proof. Let v_1 be the first vertex of X. Suppose that v_1 is not a source of X. By definition of (not being a) source vertex, there exists $(u,v_1) \in E$. Then by the definition of linearization, we know that u will be before v_1 in X, contradicting to the fact that v_1 is the first element of X. The other side can be proved symmetrically. □

We can use above algorithm to constructively prove following statement.

Claim 3. Let $G = (V, E)$ be a DAG. A vertex $u \in V$ is a source if and only if there exists a linearization X of G such that u is the first vertex in X.

Proof. We first prove that, if u is a source, then there exists a linearization where u is the first vertex of X. We prove it by showing that, we can construct such a linearization X. We can use above algorithm, and in its first step, we simply pick u. The correctness of above algorithm explains the rest. The other side is exactly Claim 2, which we have proved. □

Connectivity of Directed Graphs

For a directed graph $G = (V, E)$, its structure of connectivity can be represented as a new directed graph, called *meta-graph*, denoted as $G_M = (V_M, E_M)$. Each of the vertices of the meta-graph corresponds to a connected component of G, and two vertices $C_i, C_j \in V_M$ are connected by edge $(C_i, C_j) \in E_M$ if and only if there exists edge $(u, v) \in E$ such that $u \in C_i$ and $v \in C_j$. An example of meta-graph is given below.

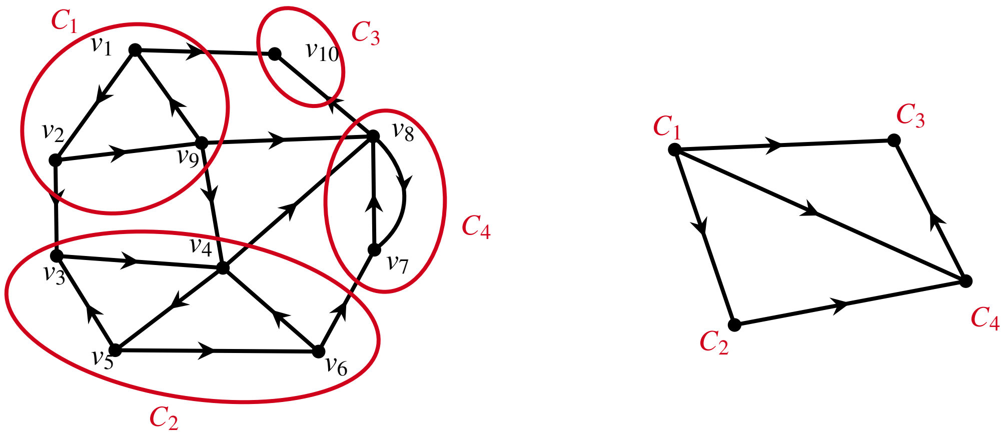


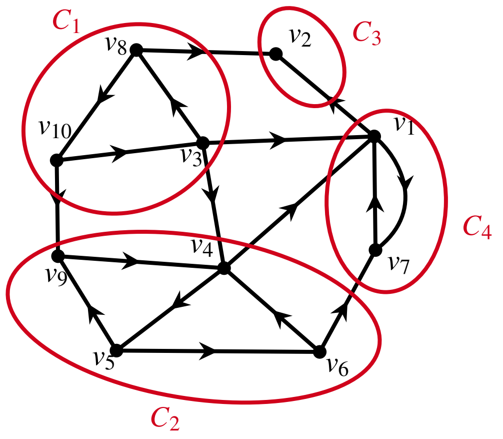
Figure 1: Example of meta-graph.

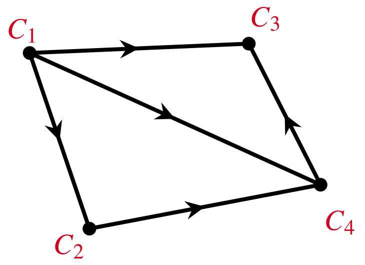
Meta-graph has an important property: it does not contain cycles, i.e., it is a directed acyclic graph (DAG).

Claim 4. The meta-graph G_M of any directed graph G is a directed acyclic graph.

Proof. Suppose conversely that G_M contains a cycle, $C_1 \to C_2 \to C_k \to C_1$, then the union of the vertices in these connected components form a single connected component, contradicting to the *maximal* property of connected component. □

We now design algorithms to determine the connected components of directed graphs and then to construct the corresponding meta-graph. Recall that the DFS algorithm introduced in Lecture A9 can successfully determine all connected components in an undirected graph. Does this work also work for directed graph? Let's try. See Figure 2. Recall that the (top-layer) of the DFS algorithm is to tranverse all vertices in some ordering, and if the current vertex v is not yet visited, we will explore v. For the example in Figure 2, let's try it with the (natrual) ordering $v_1, v_2, \cdots, v_{10}$. The resulting visited array is given in the figure (please make sure you try it yourself). Unfortunately, it does not give all connected components. In fact, the vertices marked as "1"s are the union of C_3 and C_4, and the vertices marked as "2"s are the union of C_1 and C_2. The reason is quite clear: we start with explore v_1, and during it all vertices reachable from v_1 will be marked as "1" in the visted array. It turns out that v_1 is in connected component C_4, and C_4 can reach C_3 in the meta-graph. Therefore, all vertices in C_4 and C_3 are reachable from v_1; consequently, the visited array is set as "1" for vertices in C_4 and C_3 during explore v_1. Similarly, we can also see why the vertices marked as "2"s are the union of C_1 and C_2. After exploring v_1, $\{v_1, v_2, v_7\}$ are visited; the next unvisited vertex is v_3 according to above natural ordering, the the algorithm will explore v_3. Again, during it all (unvisited) vertices reachable from v_3 will be marked as "2" in the visted array. It turns out that v_3 is in connected component C_1, and C_1 can reach C_2 in the meta-graph. Therefore, all vertices in C_1 and C_2 are reachable from v_2; consequently, the visited array is set as "2" for vertices in C_1 and C_2.





final array of *visited* after running DFS with ordering $(v_1, v_2, \cdots, v_{10})$:

v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
1	1	2	2	2	2	1	2	2	2

final array of *visited* after running DFS with a special ordering $(v_2, v_7, v_4, v_6, v_8, v_1, v_3, v_9, v_{10}, v_5)$:

v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
2	1	4	3	3	3	2	4	3	4

Figure 2: Run the DFS algorithm with an arbitrary ordering (introduced in Lecture A9) and a special ordering (pseudo-code given below) on above example.

How to fix this issue? The idea is to use a *special ordering* of vertices, instead of an arbitrary ordering, in above DFS. This special ordering should allow us to determine connected component one by one. Hence, the first vertex we explore in DFS, should be one vertex in a *sink* component of the meta-graph. In Figure 2, the only sink component is C_3, and since there is only one vertex in C_3, we should start with exploring v_2. Clearly, exploring v_2 will exactly determine the single connected component C_2. Next, after C_3 is determined, the next component we can determine is again a sink component after removing C_3 from the meta-graph. It is C_4. So, the next vertex the DFS should explore must be v_1 or v_7. It does matter which one we pick; let's assume we pick v_7 (and it does not matter where v_1 is in the ordering as long as it is after v_2 and v_7). Clearly, exploring v_2 will exactly determine the single connected component C_4—see the visited array. The next component we can determine is again a sink component after removing C_3 and C_4 from the meta-graph. It is C_2. So, the next vertex the DFS should explore must be in $\{v_4, v_5, v_6, v_9\}$. And it does matter which one we pick neither the ordering of remaining ones as long as they are behind the one we pick. Let's say we pick v_4. Clearly, exploring v_4 will exactly determine the single connected component C_2. Now the only component remaining is C_1 after removing C_2, C_3 and C_4 from the meta-graph. So, the next vertex the DFS should explore must be in $\{v_3, v_8, v_{10}\}$. Let's say we pick v_8. Clearly, exploring v_8 will exactly determine the last connected component C_1.

To abtract above observation, the determined connected components with above DFS forms a reverse-linearization of the meta-graph. Therefore, the special ordering of vertices should satisfy this condition: *the ordering of connected components sorted by their first appearance in the special ordering of vertices should form a reverse-linearization of the meta-graph.* And if the special ordering of vertices satisfies this condition, the DFS algorithm will work—it will identify all connected components.

Again see Figure 2, the ordering $(v_2, v_7, v_4, v_6, v_8, v_1, v_3, v_9, v_{10}, v_5)$ satisfies above condition. To see that, the corresponding list of connected components is $(C_3, C_4, C_2, C_2, C_1, C_4, C_1, C_2, C_1, C_2)$, and hence the ordering of their first appearance is (C_3, C_4, C_2, C_1) which is indeed a reviers-linearization of the meta-graph.

```
function DFS ($G = (V, E)$)
    num-cc = 0;
    for $v_i$ in a specific order
        if ($visited[i] = 0$)
            num-cc = num-cc + 1;
            explore ($G, v_j$);
        end if;
    end for;
end algorithm;
```

```
function explore ($G = (V, E), v_i \in V$)
    $visited[i]$ = num-cc;
    for any edge $(v_i, v_j) \in E$
        if ($visited[j] = 0$): explore ($G, v_j$);
    end for;
end algorithm;
```

To summarize, the algorithm to identify connected components of directed graphs is essentially the same with that for undirected graphs (introduced in Lecture A9, copied above), except a single line of difference (marked blue): for directed graphs, we need to explore vertices in a specific order that satisfy above condition, while for undirected graphs, we can explore all vertices in any arbitrary order.

How to find an ordering of vertices that satisfy above condition? We need a combination of two techniques: DFS-with-timing and reverse-graph.

DFS with Timing

The DFS-with-timing is a variant of DFS, which uses the following data structures (we assume $n = |V|$):

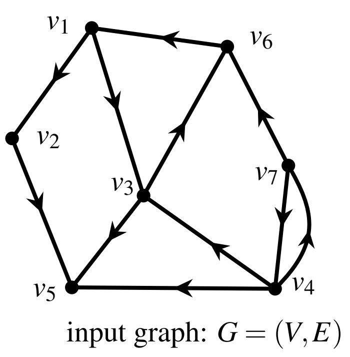
1. variable clock servers as a timer that stores the current time;
2. binary array $visited[1..n]$, where $visited[i]$ indicates if $v[i]$ has been explored, $1 \leq i \leq n$;
3. array $pre[1..n]$, where $pre[i]$ records the time of starting exploring v_i, $1 \leq i \leq n$;
4. array $post[1..n]$, where $post[i]$ records the time of finishing exploring v_i, $1 \leq i \leq n$;
5. array $postlist$, stores the vertices in decreasing order of $post[\cdot]$.

The pseudo-code of DFS with timing is given below.

```
function DFS-with-timing ($G = (V,E)$)
    $clock = 1$;
    for $i = 1 \to |V|$
        if ($visited[i] = 0$): explore ($G, v_i$);
    end for;
end algorithm;

function explore ($G = (V,E), v_i \in V$)
    $visited[i] = 1$;
    $pre[i] = clock$;
    $clock++$;
    for any edge $(v_i, v_j) \in E$
        if ($visited[j] = 0$): explore ($G, v_j$);
    end for;
    $post[i] = clock$;
    $clock++$;
    add $v_i$ to the front of $postlist$;
end algorithm;
```

An example of running DFS with timing is given below. Notice that the DFS searching partitions all edges into two categories: solid edges (u,v) implies that v is visited for the first time (and therefore explore v will start right now), while dashed edges (u,v) implies that at that time v has been visited already (and therefore v will be skipped and the next adjacent vertex of u will be examined in the for-loop).



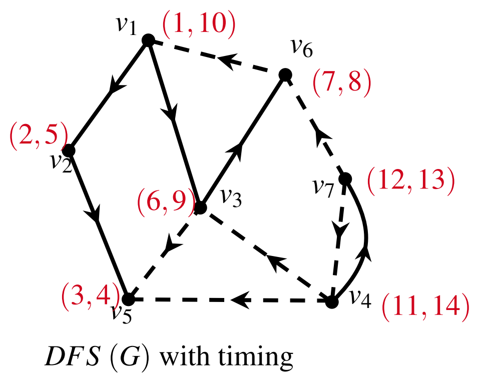
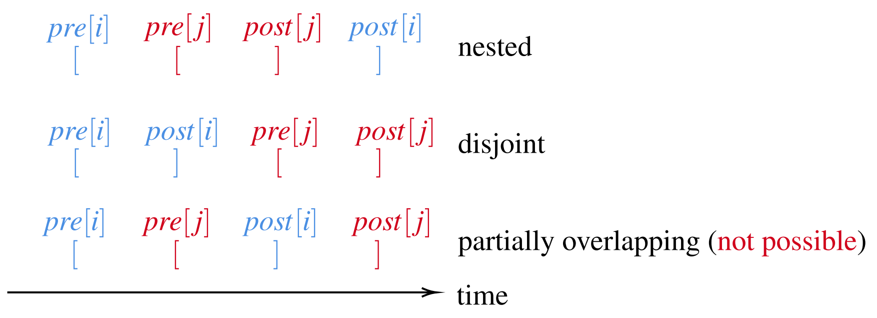


Figure 3: Example of running DFS (with timing) on a directed graph. The $[pre, post]$ interval for each vertex is marked next to each vertex. The $postlist$ for this run is $postlist = (v_4, v_7, v_1, v_3, v_6, v_2, v_5)$.

DFS with Timing and Properties

Recall that the DFS-with-timing algorithm gives an interval $[pre, post]$ for each vertex. For two vertices $v_i, v_j \in V$, their corresponding intervals can either be *disjoint*, i.e., the two intervals do not overlap, or *nested*, i.e., one interval is within the other. See Figure 1. But two intervals cannot be *partially overlapping*. Why? This is because the explore funtion is recursive. There are only two possiblities that $pre[i] < pre[j]$. The first one is that explore v_j is *within* explore v_i; in this case the recursive behaviour of explore leads to that $post[j] < post[i]$, as explore v_j must return/terminate first and then explore v_i will return/terminate. This case corresponds to that the two intervals are nested. The second one is that explore v_j starts after explore v_i finishes; this case corresponds to that the two intervals are disjoint.



$pre[i]$ $pre[j]$ $post[j]$ $post[i]$
nested
$pre[i]$ $post[i]$ $pre[j]$ $post[j]$
disjoint
$pre[i]$ $pre[j]$ $post[i]$ $post[j]$
partially overlapping (not possible)
time

Figure 1: Relations between two $[pre, post]$ intervals.

We now show and prove a key claim that relates the post values and the meta-graph.

Claim 1. Let C_i and C_j be two connected components of directed graph $G = (V, E)$, i.e., C_i and C_j are two vertices in its coresponding meta-graph $G_M = (V_M, E_M)$. If we have $(C_i, C_j) \in E_M$ then we must have that $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$.

Intuitively, following an edge in the meta-graph, the largest post value decreases. Before seeing a formal proof, please see an example in Figure 2: the largest post values for C_1, C_2, C_3, and C_4 are 9, 6, 10, and 14, and you may verify that following any edge in the meta-graph, the largest post value always decreases.

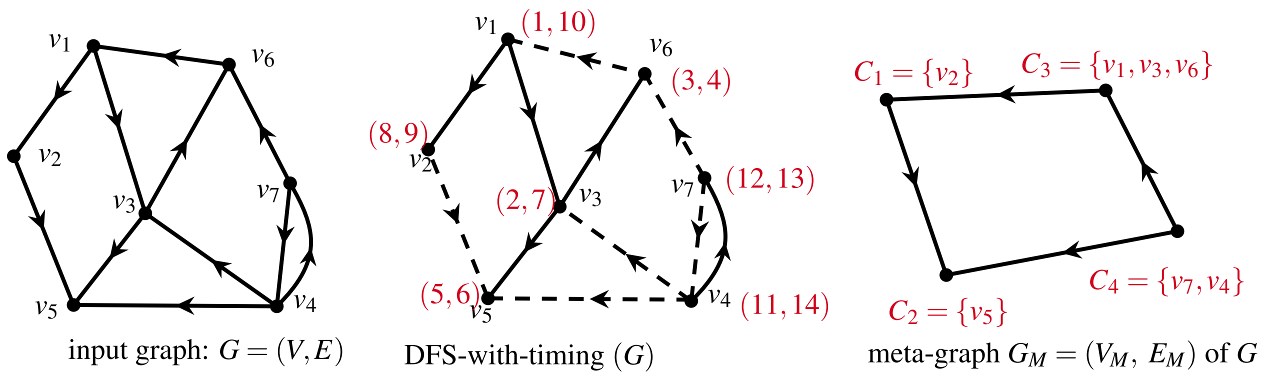


Figure 2: Example of running DFS (with timing) on a directed graph. The $[pre, post]$ interval for each vertex is marked next to each vertex. The *postlist* for this run is $postlist = (v_4, v_7, v_1, v_2, v_3, v_5, v_6)$.

Proof. Let $u^* := \arg\min_{u \in C_i \cup C_j} pre[u]$, i.e., u^* is the first explored vertex in $C_i \cup C_j$. Consider the two cases.

First, assume that $u^* \in C_i$. Then u^* can reach all vertices in $C_i \cup C_j \setminus \{u^*\}$. Hence, all vertices in $C_i \cup C_j \setminus$

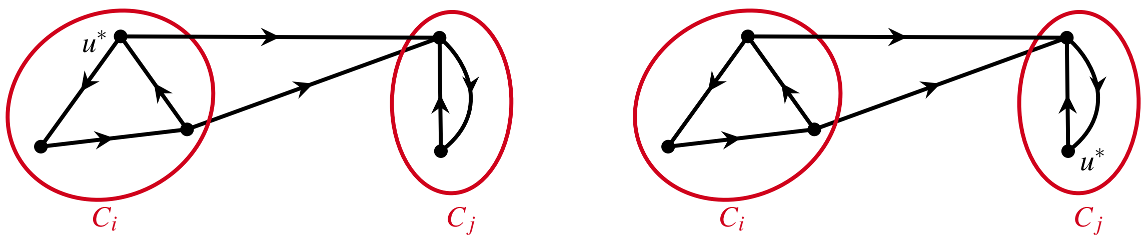


Figure 3: Two cases in proving above claim.

$\{u^*\}$ will be explored *within* exploring u^*. In other words, for any vertex $v \in C_i \cup C_j \setminus \{u^*\}$, the interval $[pre[v], post[v]]$ is a subset of $[pre[u^*], post[u^*]]$. This results in two facts: $\max_{u \in C_i} post[u] = post[u^*]$ and $\max_{v \in C_j} post[v] < post[u^*]$. Combined, we have that $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$.

Second, assume that $u^* \in C_j$. Then u^* can *not* reach any vertex in C_i; otherwise $C_i \cup C_j$ form a single connected component, conflicting to the fact that any connected component must be maximal. Hence, all vertices in C_i will remain unexplored after exploring u^*. In other words, for any vertex $v \in C_i$, the interval $[pre[v], post[v]]$ locates after (and disjoint with) $[pre[u^*], post[u^*]]$. This gives that $\max_{u \in C_i} post[u] > post[u^*]$. Besides, we also have $\max_{v \in C_j} post[v] = post[u^*]$ as all vertices in $C_j \setminus \{u^*\}$ will be examined within exploring u^*. Combined, we have that $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$. □

The above key claim directly suggest a property about the structure of the meta-graph, given below.

Corollary 1. The list of all connected components in V_M sorted in decreasing order of $\max_{u \in C_i} post[u]$, $1 \le i \le k$, is a linearization of G_M.

Verify this is the case in Figure 2. Answer: such list is (C_4, C_3, C_1, C_2), and it is a linearization of G_M.

Above property also suggests us to consider an *ordering of vertices* in descending post values. Specifically, we define *postlist* as such ordering of vertices. Notice that postlist can be stored in an array and can be efficiently constructed in DFS-with-time. Please refer to Lecture A10 for its pseudo-code. See Figure 2 for an example.

As a direct consequence of Corollary 1 and the definition of postlist, we have the following property. This is true as "the first appearance" exactly gives the "largest post value" since the postlist is sorted in descending order of post values.

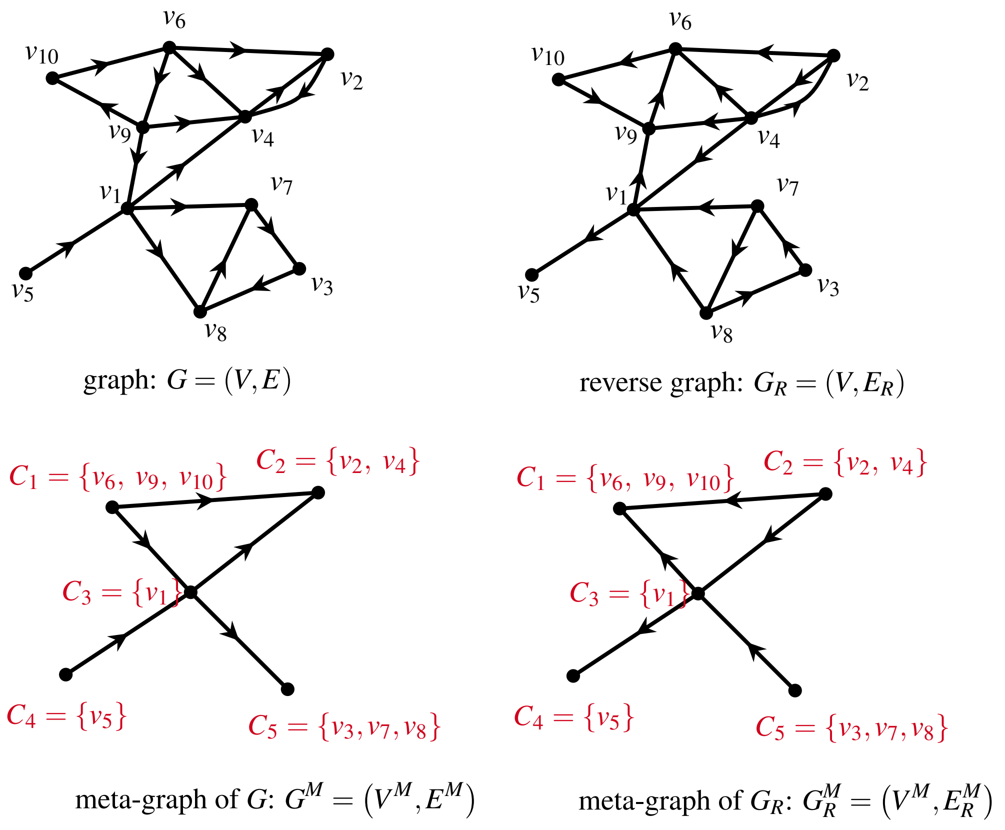
Corollary 2. The list of all connected components in V_M sorted by their first appearance of in the postlist is a linearization of G_M.

Verify this is the case in Figure 2. Answer: following $postlist$ the corresponding list of connected components is $(C_4, C_4, C_3, C_1, C_3, C_2, C_3)$. The first appearance of each component is (C_4, C_3, C_1, C_2), which is a linearization of G_M and exactly the one that is sorted by their largest post value.

To summarize, now we can build a postlist that satisfies above Corollary 2. This almost achieves the desired property of the special ordering: "the ordering of connected components sorted by their first appearance in the special ordering should form a reverse-linearization of the meta-graph." The only difference is that postlist implies a *linearization* of the meta-graph while special ordering requires a *reverse-linearization* of the meta-graph. Below, we introduce *reverse graph* to fill this gap.

Reverse Graph

Definition 1. Let $G = (V, E)$ be a directed graph. The *reverse graph* of G, denoted as $G_R = (V, E_R)$, has the same set of vertices and edges with reversed direction, i.e., $(u, v) \in E$ if and only if $(v, u) \in E_R$.



v_{10}
v_6
v_2
v_9
v_4
v_1
v_7
v_3
v_5
v_8
graph: $G = (V, E)$
reverse graph: $G_R = (V, E_R)$
$C_1 = \{v_6, v_9, v_{10}\}$
$C_2 = \{v_2, v_4\}$
$C_3 = \{v_1\}$
$C_4 = \{v_5\}$
$C_5 = \{v_3, v_7, v_8\}$
meta-graph of G: $G^M = (V^M, E^M)$
meta-graph of G_R: $G_R^M = (V^M, E_R^M)$

Figure 4: Graph and its reverse graph, and the corresponding meta-graphs.

Following properties can be easily proved using above definition.

Property 1. There is a path from u to v in G if and only if there is a path from v to u in G_R. In other words, u can reach v in G if and only if u can be reached from v in G_R.

Property 2. $(G_R)_R = G$.

Property 3. A vertex v of DAG G is a source vertex if and only if v is a sink vertex of G_R.

Property 4. G and G_R has the same set of connected components.

Property 5. The meta-graph of G_R is the reverse graph of the meta-graph of G. Formally, $(G_R)^M = (G^M)_R$.

Property 6. X is a linearization of DAG G if and only if the reverse of X is a linearization of G_R (or, X is a reverse-linearization of G_R). In particular, since any meta-graph is a DAG, we have that X is linearization of G^M if and only if X is a reverse-linearization of $(G^M)_R = (G_R)^M$, and that X is a linearization of $(G_R)^M = (G^M)_R$ if and only if X is a reverse-linearization of G^M.

Algorithm to Determine Connected Components in Directed Graphs

Combining above DFS-with-timing and reverse graph, here is the pseudo-code we use to obtain a special ordering. The key is that we run DFS-with-timing on the reverse graph G_R, instead of on the given graph G.

```
Algorithm to determine the specific order
    build $G_R$ of $G$;
    run DFS with timing on $G_R$ to get $postlist$;
    return $postlist$;
end algorithm;
```

We now show that, above *postlist*, i.e., the *postlist* obtained by running DFS-with-timing on G_R, satisfies the condition that "the ordering of connected components sorted by their first appearance in above *postlist* forms a reverse-linearization of the meta-graph G_M". In fact, since above *postlist* is obtained by running DFS-with-timing on G_R, according to Corollary 2, the list of all connected components sorted by their first appearance of in above *postlist* is a linearization of $(G_R)^M$. According to Property 6 of reverse graph, we know that a linearization of $(G_R)^M$ is a reverse-linearization of G_M.

Above *postlist* can then be used by the DFS algorithm as a special order to obtain all connected components, with pseudo-code copied below.

```
function DFS $(G = (V, E))$
    num-cc = 0;
    for $v_i$ in the $postlist$ obtained above
        if $(visited[i] = 0)$
            num-cc = num-cc + 1;
            explore $(G, v_j)$;
        end if;
    end for;
end algorithm;
```

```
function explore $(G = (V, E), v_i \in V)$
    $visited[i] =$ num-cc;
    for any edge $(v_i, v_j) \in E$
        if $(visited[j] = 0)$: explore $(G, v_j)$;
    end for;
end algorithm;
```

The entire algorithm runs in $\Theta(|V| + |E|)$ time.

Deciding Directed Acyclic Graphs

How to decide if a given directed graph is DAG or not? The following variant of DFS (with timing) gives an algorithm. Specifically, when the algorithm examines an edge (v_i, v_j): if v_j has been explored *and* its post-number hasn't been set yet, then the algorithm reports that G is not a DAG.

```
function DFS ($G = (V,E)$)
    $clock = 1$;
    $visited[i] = 0, pre[i] = -1, post[i] = -1$, for $1 \leq i \leq |V|$;
    for $i = 1 \to |V|$
        if ($visited[i] = 0$)
            explore ($G, v_j$);
        end if;
    end for;
end algorithm;
```

```
function explore ($G = (V,E), v_i \in V$)
    $visited[i] = 1$;
    $pre[i] = clock$;
    $clock{+}{+}$;
    for any edge $(v_i, v_j) \in E$
        if ($visited[j] = 0$): explore ($G, v_j$);
        else if ($post[j] = -1$): report "$G$ is not a DAG";
    end for;
    $post[i] = clock$;
    $clock{+}{+}$;
end algorithm;
```

Now let's prove this algorithm is correct. We first prove that if G is not a DAG then the algorithm will always give that report at some time. Assume that G contains a cycle C as it is not a DAG. Let $v_j \in C$ be the first vertex that is explored in C. Let $(v_i, v_j) \in E$ be an edge in C. As v_j can reach v_i, within exploring v_j there will be a time that v_i will be explored. Consider the time of examining edge (v_i, v_j) within exploring v_i: at this time $visited[j]$ has been set as 1, but its post-number hasn't been set, as now the algorithm is still within exploring v_j. Therefore, the algorithm will report that G dis not a DAG.

We then prove that if the algorithm gives that report then G indeed is not a DAG. Consider that the algorithm is exploring v_i, examining edge (v_i, v_j) and finds $visited[j] = 1$ *and* $post[j] = -1$. The fact that $post[j]$ hasn't been set implies that the algorithm is within exploring v_j. So we have that v_j can reach v_i, as now we are exploring v_i. In addition, there exists edge (v_i, v_j). Combined, G contains cycle.

Note that this algorithm is essentially determining if there exists edge $(v_i, v_j) \in E$ such that the interval $[pre[i], post[i]]$ is within interval $[pre[j], post[j]]$. (Such edges are called *back edges* in textbook [DPV], page 95.)

Finding a Linearization of a DAG

DFS-with-timing can also be used to find a linearization of a DAG: we simply run DFS-with-timing on the given DAG G and get the postlist (the list of vertices in decreasing order of post value); the postlist will be a linearization of G.

Why? First, observe that each connected component of a DAG G contains exactly one vertex, i.e., each vertex in a DAG G forms the connected component of its own. (Can you spot this using Figure 1?) This is because, if a connected component contains at least two vertices u and v then u can reach v and v can reach u so a cycle must exist. Consequently, the meta-graph G_M of any DAG G is also itself, i.e., $G = G_M$.

Now let's interpret the conclusions for general directed graphs we obtained in Lecture A11 in the context of DAGs. Consider Claim 1 in Lecture A11: *Let C_i and C_j be two connected components of directed graph $G = (V,E)$, i.e., C_i and C_j are two vertices in its coresponding meta-graph $G_M = (V_M, E_M)$. If we have $(C_i, C_j) \in E_M$ then we must have that* $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$. As in a DAG, components C_i and C_j will degenerate into two vertices, say v_i and v_j, and its meta-graph G_M is the same as G, we can translate this claim for DAG as: if in a DAG we have $(v_i, v_j) \in E$ then we must have $post[i] > post[j]$. This immediately gives the desired conclusion following the definition of linearization and the definition of postlist: the postlist is a linearization of G.

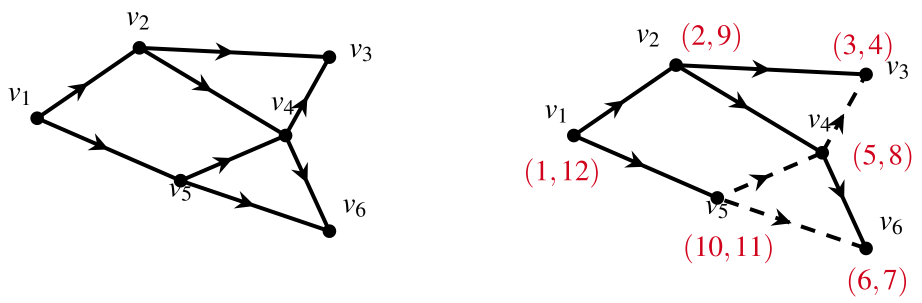


Figure 1: Example of running DFS (with timing) on a DAG G. The $[pre, post]$ interval for each vertex is marked next to each vertex. The *postlist* for this run is $(v_1, v_5, v_2, v_4, v_6, v_3)$, which is a linearization of G.

Queue

A *queue* data structure supports the following four operations.

1. empty (Q): decides if queue Q is empty;
2. insert (Q, x): add element x to Q;
3. find-earliest (Q): return the earliest added element in Q;
4. delete-earliest (Q): remove the earliest added element in Q.

To implement above operations, we can use a (dynamic) array S to stores all elements, and use two pointers, *head* and *tail*, where *head* pointer always points to the first available space in S, and *tail* pointer always points to the earliest added element in S. When we add an element to S we can directly add it to the place *head* points to, and when we delete the earliest added element, we can directly remove the one *tail* points to.

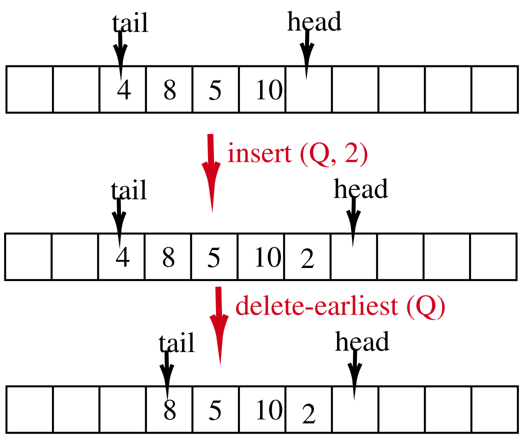


Figure 2: An example of queue.

```
function empty($Q$)
    if $head = tail$: return true;
    else: return false;
end function;

function insert($Q$, $x$)
    $S[head] = x$;
    head = head + 1;
end function;

function find-earliest($Q$)
    return $S[tail]$;
end function;

function delete-earliest($Q$)
    tail = tail + 1;
end function;
```

Note, a queue data structure exhibits a first-in-first-out property (while a stack is first-in-last-out).

Priority Queue

In a priority queue, each element is associated with a *priority* (also called key). In other words, each element in a priority queue is a pair (*key, value*), where *key* indicates its priority, while *value* stores the actual data. A *priority queue* data structure supports the following operations.

1. empty (PQ): decides if priority queue PQ is empty;
2. insert (PQ, x): add element x to PQ;
3. find-min (PQ): return the element in PQ with smallest key (i.e., highest priority);
4. delete-min (PQ): remove the element in PQ with smallest key (i.e., highest priority).

5. decrease-key (PQ, pointer-to-an-element, new-key): set the key of the specified element as the given new-key.

Note that a queue can be regarded as a special case of priority, for which the priority is the time an element is added to the queue.

There are numerous different implementations for priority queue (check wikipedia). Here we introduce one of them, *binary heap*. To do it, let's first formally introduce *heap*.

A *heap* is a (rooted) tree data structure satisfies the *heap property*. A heap is either a *min-heap* if it satisfies the *min-heap property*: for any edge (u,v) in the (rooted) tree T, the key of u is smaller than that of v, or a *max-heap* if it satisfies the *max-heap property*: for any edge (u,v) in the (rooted) tree T, the key of u is larger than that of v. Here we always consider a heap as a min-heap, unless otherwise specified.

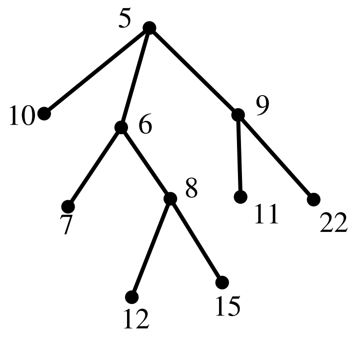


Figure 3: An example of heap. The key of an element is next to vertices

A *binary heap* is a heap with the tree being the *complete binary tree*. A *complete binary tree* is a binary tree (i.e., each vertex has at most 2 children) and that in every layer of the tree, except possibly the last, is completely filled, and all vertices in the last layer are placed from left to right.

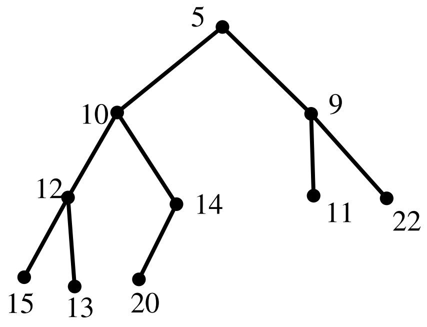


Figure 4: An example of binary heap.

Since a binary heap T is so regular, we can use an array S to store its elements (rather than using adjacency list). The root (i.e., layer 0) of T is placed in $S[1]$ (we assume that the index of S starts from 1), the first element of the layer 1 is placed in $S[2]$, and so on. Generally, the j-th element in the i-th layer of T will be placed in $S[2^i + j - 1]$. We can also easily access the parent and children of an element:

1. the parent element of $S[k]$ is $S[k/2]$ (floor of $k/2$);
2. the left child of $S[k]$ is $S[2k]$; the right child of $S[k]$ is $S[2k+1]$.

We now introduce two common procedures used in implementing a binary heap. These procedures apply when one vertex violates the heap property, and they can adjust the heap to make it satisfy the heap property.

The *bubble-up* function applies when a vertex has a smaller key than its parent.

```
function bubble-up ($S$, $k$)
    $p = k/2$;
    if ($S[k].key < S[p].key$);
        swap $S[p]$ and $S[k]$;
        bubble-up ($S$, $p$);
    end if;
end function;
```

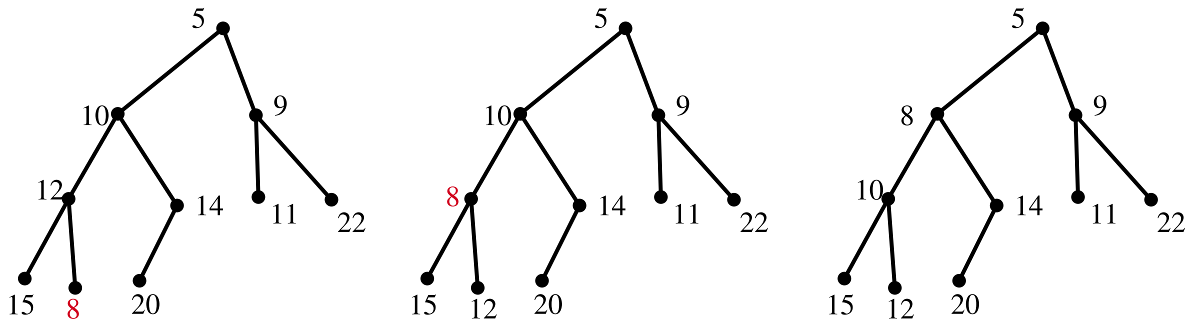


Figure 5: Illustrating bubble-up procedure.

The *sift-down* function applies when a vertex has a larger key than its children.

```
function sift-down ($S$, $k$)
    $c = \arg\min_{t\in\{2k,2k+1\}} S[t].key$ be the index of the child of $S[k]$ with smaller key;
    if ($S[k].key > S[c].key$);
        swap $S[c]$ and $S[k]$;
        sift-down ($S$, $c$);
    end if;
end function;
```

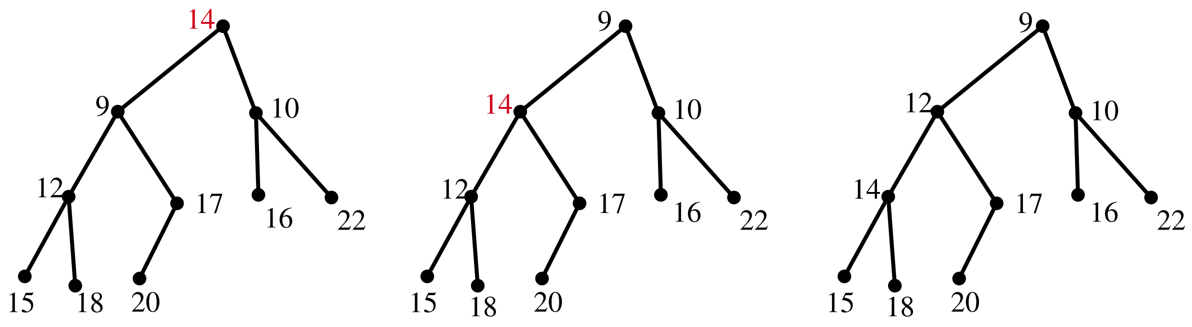


Figure 6: Illustrating sift-down procedure.

Dijkstra's Algorithm (continued)

We rewrite the formula in Corollary 1 (last Lecture) into an equivalent form by separating the minimization into two levels:

$$distance(s, v^*_{k+1}) = \min_{v \in V \setminus R_k} \min_{u \in R_k, (u,v) \in E} (distance(s,u) + l(u,v)).$$

Now we define the inner level of minimization with a new term $dist_k(v)$:

$$dist_k(v) := \min_{u \in R_k, (u,v) \in E} (distance(s,u) + l(u,v)).$$

Then we have

$$distance(s, v^*_{k+1}) = \min_{v \in V \setminus R_k} dist_k(v).$$

Intuitively, $dist_k(v)$ gives the length of the shortest path from s to v by only using vertices in R_k.

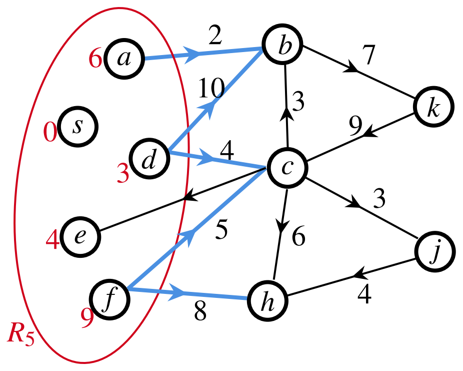


Figure 1: Try above formulas in Figure 1. Answer: $dist_5(b) = \min\{8, 13\} = 8$, $dist_5(c) = \min\{7, 14\} = 7$, $dist_5(h) = 17$, $dist_5(k) = \infty$, $dist_5(j) = \infty$; hence $v^*_6 = c$ and $distance(s, v^*_6) = dist_5(c) = 7$.

With $dist_k$ available, the next closest vertex, i.e., v^*_{k+1} can be easily calculated by picking the one with smallest $dist_k$ value. More importantly, we can design a more efficient procedure to calculate $dist_{k+1}$ in the next iteration by largely reusing $dist_k$. To see that, recall its definition, for any $v \in V \setminus R_{k+1}$, we have

$$dist_{k+1}(v) = \min_{u \in R_{k+1}, (u,v) \in E} (distance(s,u) + l(u,v)).$$

Note that $R_{k+1} = R_k \cup \{v^*_{k+1}\}$. Hence

$$dist_{k+1}(v) = \begin{cases} \min\{dist_k(v), distance(s, v^*_{k+1}) + l(v^*_{k+1}, v)\} & \text{if} \quad (v^*_{k+1}, v) \in E \\ dist_k(v) & \text{if} \quad (v^*_{k+1}, v) \notin E \end{cases}$$

In other words, when calculating $dist_{k+1}$, we only need to examine the out-edges of v^*_{k+1} and update only if the use of v^*_{k+1} leads to a shorter path. The pseudo-code of calculating $dist_{k+1}$ from $dist_k$ is given below. Before calling this updating procedure, $dist_{k+1}$ is first copied from $dist_k$.

```
procedure update-dist ($dist_k, v^*_{k+1}$)
    for $(v^*_{k+1}, v) \in E$
        if ($dist_k(v) > distance(s, v^*_{k+1}) + l(v^*_{k+1}, v)$)
            $dist_{k+1}(v) = distance(s, v^*_{k+1}) + l(v^*_{k+1}, v)$;
        end if;
    end for;
end procedure;
```

Try above procedure with the example below.

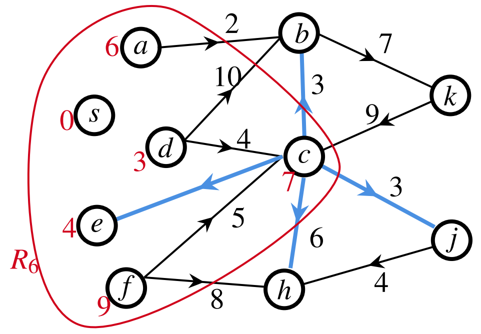


Figure 2: Following Figure 1, we have that $v_6^* = c$. We now want to calculate $dist_6$ using $dist_5$. We consider the out-edges of c, marked as thick blue edges. Eventually, $dist_6(b) = 8$, $dist_6(h) = 13$, $dist_6(k) = \infty$, and $dist_6(j) = 10$.

The above procedure enables fast calculation of $dist_k$. The last piece of Dijkstra's algorithm comes with the use of *priority queue* to quickly pick the next closest vertex, i.e., to calculate $v_{k+1}^* = \arg\min_{v \in V \setminus R_k} dist_k(v)$. To this end, the priority queue PQ always stores $V \setminus R_k$, and for each vertex v that is stored in PQ, its priority is $dist_k(v)$. In this way, every time we call find-min (PQ), it gives us $\min_{v \in V \setminus R_k} dist_k(v)$. In the complete Dijkstra's algorithm, we don't need to implicitly maintain R_k, as PQ is always complement to R_k. In order to maintain this invariant, we delete v_{k+1}^* from PQ, by calling delete-min, at the time of adding v_{k+1}^* to R_{k+1}. In order to guarantee that the priority of v is always $dist_k(v)$, we call decrease-key every time we update $dist_k$ of a vertex.

```
Algorithm Dijkstra ($G = (V, E)$, $l(e)$ for any $e \in E$, $s \in V$)
    $dist[v] = \infty$, for any $v \in V$;
    init an empty priority queue $PQ$;
    for any $v \in V$: insert ($PQ$, $v$), where the priority of $v$ is $\infty$;
    $dist[s] = 0$;
    decrease-key ($PQ, s, 0$);
    while (empty ($PQ$) = false)
        $u$ = find-min ($PQ$);
        delete-min ($PQ$);
        for each edge $(u, v) \in E$
            if ($dist[v] > dist[u] + l(u, v)$)
                $dist[v] = dist[u] + l(u, v)$;
                decrease-key ($PQ$, $v$, $dist[v]$);
            end if;
        end for;
    end while;
end algorithm;
```

The pseudo-code for complete Dijkstra's algorithm is given above. We use array *dist* of size n to store $dist_k$. Where are the distances for those vertices in R_k (i.e., those are not in PQ) stored? They are in array *dist* as well. This is because, at the time v^*_{k+1} is identified and added to R_{k+1} (i.e., removed from *PQ*), $dist_k$ value for this vertex is exactly its distance. In fact, at any time of the algorithm, $dist[v] = distance(s,v)$ for any vertex v that is not in *PQ*. Finally, the algorithm don't explicitly maintain an index k: this index implicitly increases in every iteration of the while loop.

Here are some facts about Dijkstra's algorithm. First, $dist[v]$ is always an upper bound of the distance.

Fact 1. At any time of the algorithm, $dist[v] \geq distance(s,v)$.

This is because, by definition, $dist_k(v)$ is the length of shortest path from s to v using only vertices in R_k. In other words, $dist_k(v)$ represents the length of the optimal path of a subset (of all possible paths from s to v).

Second, once v is removed from *PQ*, $dist[v]$ won't change and remain as $distance(s,v)$.

Fact 2. At any time of the algorithm, if v is not in *PQ*, then $dist[v] = distance(s,v)$.

This is because, at the time v is deleted from *PQ*, $dist[v] = distance(s,v)$. Following Fact 1, it will remain this minimized value. Notice though, in Dijkstra's algorithm, a vertex v not in *PQ* might be "touched" by an edge (v^*_{k+1}, v), where v^*_{k+1} is the newly determined closest vertex, but the actual update will not happen. See an example in Figure 2 of Lecture 17, edge (c,e).

The running time of Dijkstra's algorithmd depends on the specific implementation of priority queue used. Consider using binary heap. The break-down of running time is given below. Note that each vertex will be picked from the *PQ* at most once and each edge will be examined at most once (for directed graph) or at most twice (for undirected graph). The total running time is $\Theta((|V| + |E|)\log|V|)$.

1. initialization: $\Theta(|V|)$;
2. insert (*PQ*): $|V| \times \Theta(\log|V|)$;
3. empty (*PQ*): $|V| \times \Theta(1)$;
4. find-min (*PQ*): $|V| \times \Theta(1)$;
5. delete-min (*PQ*): $|V| \times \Theta(\log|V|)$;
6. updating-dist: $|E| \times \Theta(1)$;
7. decrease-key (*PQ*): $|E| \times \Theta(\log|V|)$;

Properties of Shortest Path Problem

To prepare to solve the shortest path problem with negative edge length, we first see some properties.

A negative cycle C in a graph is a cycle with negative length, i.e., $l(C) := \sum_{e \in C} l(e) < 0$. In the presence of negative cycle, if we don't limit the number of edges in a path, then the length of a path could goes to negative infinity. In other words, the shortest path may not exist. Therefore, in a graph with negative edge length, we want to detect if there exists negative cycle. We will show an algorithm (the Bellman-Ford algorithm) can be used to detect negative cycles.

A path p in a graph is *simple* if p does not have repeating vertices. If a graph G does not contain negative cycle, then for any pair of vertices u and v, if u can reach v, then there always exists a simple shortest path from u to v, as otherwise we can skip the cycle in it to get a better or same-length path. If all cycles in graph G are positive then every shortest path is simple.

Property 1. If G does not contain negative cycles, for every $u, v \in V$, there exists a shortes path from u to v with at most $(|V| - 1)$ edges.

Shortest path admits the following *optimal substructure* property. Intuitively, this property states that, the shortest path from u to v contains the shortest path from u to any internal vertex on this path (formally described below). Essentially, this is why shortest path problem can be solved efficiently.

Property 2. Let $p = (v_1, v_2) \to (v_2, v_3) \to \cdots \to (v_{k-1}, v_k)$ be the shortest path from v_1 to v_k. Then for any $1 \le i \le k$, $p_i := (v_1, v_2) \to (v_2, v_3) \to \cdots \to (v_{i-1}, v_i)$, i.e., the portion of p from v_1 to v_i, is the shortest path from v_1 to v_i.

Proof. Suppose that $p_i = (v_1, v_2) \to (v_2, v_3) \to \cdots \to (v_{i-1}, v_i)$ is not the shortest path from v_1 to v_i. Assume that q is the shorest path from v_1 to v_i. Then we can construct a path from v_1 to v_k shorter than p, by concatenating q and $(v_i, v_{i+1}) \to \cdots \to (v_{k-1}, v_k)$. This contradicts to the fact that p is the shortest path from v_1 to v_k. $\square$

Note that the above property holds even for graphs with negative edge length (as in the proof we don't assume anything about edge length). This property also immediately implies the following fact.

Property 3. If we know that there exists one shortest path from s to v such that (w, v) is the last edge on this shortest path, then we have that $distanct(s, v) = distance(s, w) + l(w, v)$.

Bellman-Ford Algorithm

Bellman-Ford algorithm can be used to solve the (single-source) shortest path problem with negative edge length, and its extension can also be used to detect if a graph contains negative cycle (reachable from the given source).

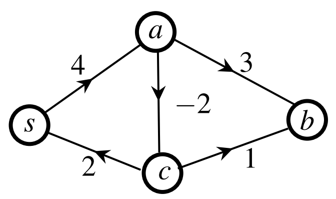
Bellman-Ford algorithm is quite simple. It only maintain an array, *dist* of size $|V|$, as its data structure. And it just does a bunch of "update" operations. An "update" function takes an edge $e = (u,v)$ as input, and updates $dist[v]$ as $dist[u] + l(u,v)$ if the former is larger than the latter.

```
procedure update(edge $(u,v) \in E$)
    if ($dist[v] > dist[u] + l(u,v)$)
        $dist[v] = dist[u] + l(u,v)$;
    end if;
end procedure;
```

Bellman-Ford algorithm iterates $(|V|-1)$ rounds, and in each round, updates *all* edges, in an arbitrary order. If the given G does contain negative cycle reachable from given s, when the algorithm terminates, we will have that $dist[v] = distance(s,v)$ for every $v \in V$.

```
Algorithm Bellman-Ford ($G = (V,E)$, $l(e)$ for any $e \in E$, $s \in V$)
    init an array $dist$ of size $|V|$;
    $dist[s] = 0$; $dist[v] = \infty$ for any $v \neq s$;
    for $k = 1 \to |V|-1$
        for each edge $(u,v) \in E$
            $update(u,v)$;
        end for;
    end for;
end algorithm;
```

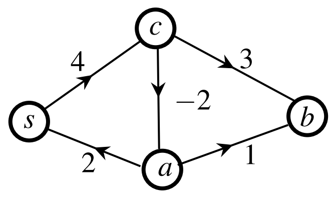
Since *update* function takes constant time, clearly, Bellman-Ford algorithm runs in $\Theta(|V| \cdot |E|)$ time. See examples below.



graph

	s	a	b	c
init	0	∞	∞	∞
$R1$	0	4	∞	∞
$R2$	0	4	3	2
$R3$	0	4	3	2

dist array



graph

	s	a	b	c
init	0	∞	∞	∞
$R1$	0	∞	∞	4
$R2$	0	2	7	4
$R3$	0	2	3	4

dist array

Figure 1: The dist array (after each round) running Bellman-Ford algorithm on each example. In each example, in each round, we choose to update all edges in lexicographic order, i.e., $(a,b),(a,c),(c,b),(c,s),(s,a)$.

Now let's see why this algorithm is correct. We first show an invariant about the data structure *dist* array:

Fact 1. Throughout the algorithm, if $dist[v] \neq \infty$ then $dist[v]$ represents the length of some path from s to v. In other words, $dist[v] \geq distance(s,v)$ throughout the algorithm, as $dist[v]$ represents the length of *some* path from s to v, while $distance(s,v)$ represents the length of the *shortest* path from s to v.

Clearly, in the initialization step which sets $dist[s] = 0$ and $dist[v] = \infty$ for all $v \neq s$, above claim holds, as $dist[s]$ stores a path from s to s without any edge and therefore its length is 0. Now to show above fact is correct throughout the algorithm, we just need to show that the "update" operation keeps this invariant (as this algorithm does nothing else but "update" operations).

Fact 2. The update operation keeps the invariant that $dist[v]$ represents the length of some path from s to v when $dist[v] \neq \infty$, i.e., $dist[v] \geq distance(s,v)$, for every $v \in V$.

Proof. We prove this by induction w.r.t. the sequence of update operations. Assume that up to the n-th update opertion above claim holds, i.e., $dist[v]$ stores the length of some path from s to v when $dist[v] \neq \infty$. Now consider the $(n+1)$-th update operation on edge $e = (u,v)$. Assume that $dist[v] > dist[u] + l(u,v)$, as otherwise this operation does not change $dist$ and the claim continues to be true. Now $dist[v]$ is updated as $dist[u] + l(u,v)$. Since, according to the inductive assumption, $dist[u]$ stores the length of some path from s to u, we have that $dist[v]$ stores the length of the path that consists of the aforementioned path from s to u followed by edge (u,v). □

In Bellman-Ford algorithm, $dist[v]$ starts from a trivial upper bound (i.e., infinity) of $distance(s,v)$, and will get closer and closer to $distance(s,v)$ through the "update" procedures, and eventually reach $distance(s,v)$. We now state the conditions for this to happen.

Fact 3. If edge (u,v) is the last edge on one shortest path from s to v and $dist[u] = distance(s,u)$, then after $update(u,v)$ we will have $dist[v] = distance(s,v)$.

Since edge (u,v) is the last edge on one shortest path from s to v, according to Property 3 of Lecture A14, we know that $distance(s,v) = distance(s,u) + l(u,v) = dist[u] + l(u,v)$. Through $update(u,v)$, $dist[v]$ will be compared with $dist[u] + l(u,v) = distance(s,v)$. The first case will be that $dist[v] \leq dist[u] + l(u,v) = distance(s,v)$. Notice that in this case we must have $dist[v] = distance(s,v)$ according to above fact, i.e., $dist[v]$ already stores the distance. The second case will be that $dist[v] > dist[u] + l(u,v) = distance(s,v)$, and in this case the "update" function will set $dist[v] = dist[u] + l(u,v) = distance(s,v)$. Hence, in either case, we will have $dist[v] = distance(s,v)$ after updating edge (u,v). □

Suppose that $s \to v_1 \to v_2 \to \cdots \to v$ is one shortest path from s to v. In the initialization step we have $dist[s] = distance(s,s) = 0$. If at a later time, $update(s,v_1)$ is executed, then following above Fact 3, we know that $dist[v_1] = distance(s,v_1)$ after this update (reasons: $dist[s] = distance(s,s)$, and (s,v_1) is the last edge on one shortest path from s to v_1 according to the optimal substrcture property). Once $dist[v_1]$ becomes $distance(s,v_1)$, $dist[v_1]$ will stay as $distance(s,v_1)$ according to Fact 1. If at a later time $update(v_1,v_2)$ happens then following Fact 3, we know that $dist[v_2] = distance(s,v_2)$. Note that it doesn't matter if additional updates happen between $update(s,v_1)$ and $update(v_1,v_2)$. We can continue this argument; a general form is summarized below.

Fact 4. If there exists a sequence of udpate procedures (not necessarily consecutive) that update all the edges following one shortest path from s to v, then after that we will have $dist[v] = distance(s,v)$. Again, there can be other "update"(s) between any two "update"s in this sequence.

But we don't know the the shortest path in advance. That's fine. As the number of edges in the shortest path will not exceed $(|V|-1)$, the Bellman-Ford algorithm simply update *all* edges in each round, and do this $(|V|-1)$ times. This therefore guarantees that the i-th edge on the shortest path can be updated during the

i-th round. Consequently, this guarantees the existence of a sequence of update procedures that update all edges following the shortest path. This analysis leads to the following conclusion, which actually proves the correctness of Bellman-Ford algorithm. See an illustration in Figure 2.

Fact 5. If G does not contain negative cycle, then we have $dist[v] = distance(s,v)$ for $v \in V$ after $|V|-1$ rounds. In particular, let p be one shortest path from s to v with k edge. Then after k rounds of the Bellman-Ford algorithm, $dist[v] = distance(s,v)$.

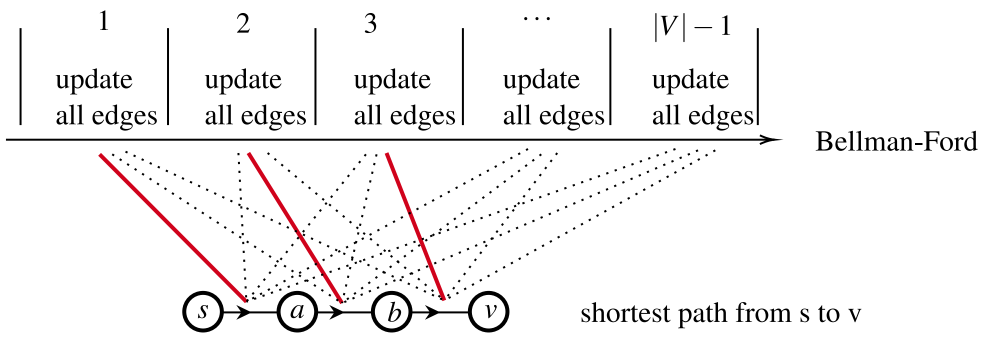


Figure 2: Illustration of the correctness of the Bellman-Ford algorithm. Dotted lines represent additional updates on the corresponding edge.

Detecting Negative Cycles

We can slightly modify Bellman-Ford algorithm to detect if a given graph contains negative cycle that is reachable from s. The algorithm does one more round of updates, in which it determines if some $dist$ value can be further reduced.

```
Algorithm Bellman-Ford-Detect-Negative-Cycle ($G = (V,E)$, $l(e)$ for any $e \in E$, $s \in V$)
    init an array $dist$ of size $|V|$;
    $dist[s] = 0$; $dist[v] = \infty$ for any $v \neq s$;
    for $k = 1 \to |V|-1$
        for each edge $(u,v) \in E$
            $update(u,v)$;
        end for;
    end for;
    for each edge $(u,v) \in E$
        if ($dist[v] > dist[u] + l(u,v)$): report $G$ contains negative cycle and exit
    end for;
    report that $G$ does not contain negative cycle
end algorithm;
```

Let's show that above algorithm is correct. We first prove that, if G does not contain negative cycle (reachable from s), then in above additional round $dist[v] > dist[u] + l(u,v)$ will never happen, i.e., we will get the report that "G does not contain negative cycle". As per Fact 5 and the assumption that G does not contain negative cycle, we know that $dist[v] = distance(s,v)$ after $|V|-1$ rounds. Also, according to Fact 2, update function will never make $dist[v]$ smaller than $distance(s,v)$ when G does not contain negative cycle. Hence, during the $|V|$-th round in above algorithm, none of the *dist* value can be further reduced.

We then prove that, if G contains negaive cycle (reachable from s), then in above additional round, there must exist an edge (u,v) such that $dist[v] > dist[u] + l(u,v)$. Suppose conversely that, in above additional round, all edges satisfy $dist[v] \leq dist[u] + l(u,v)$. Let $C = v_1 \to v_2 \to \cdots \to v_{k-1} \to v_k \to v_1$ be one negative reachable from s. We have $\sum_{e \in C} l(e) < 0$ as C is a nagative cycle. Applying $dist[v] \leq dist[u] + l(u,v)$ to all edges in C gives:

$$\begin{array}{rcl} dist[v_2] & \leq & dist[v_1] + l(v_1, v_2) \\ dist[v_3] & \leq & dist[v_2] + l(v_2, v_3) \\ & \cdots & \\ dist[v_k] & \leq & dist[v_{k-1}] + l(v_{k-1}, v_k) \\ dist[v_1] & \leq & dist[v_k] + l(v_k, v_1) \end{array}$$

Summing up both sides of all above inequalities gives $\sum_{e \in C} l(e) \geq 0$, a contradiction.

CMPSC 465 Data Structures & Algorithms
Spring 2022 Chunhao Wang and Mingfu Shao

Solution 01

1. (0 pts.) Acknowledgements. The assignment will receive a 0 if this question is not answered.

1. If you worked in a group, list the members of the group. Otherwise, write "I did not work in a group."
2. If you received significant ideas about your solutions from anyone not in your group, list their names here. Otherwise, write "I did not consult anyone except my group members".
3. List any resources besides the course material that you consulted in order to solve the material. If you did not consult anything, write "I did not consult any non-class materials."

2. (15 pts.) For each pairs of functions below, indicate one of the three: $f = O(g)$, $f = \Omega(g)$, or $f = \Theta(g)$.

1. $f(n) = n^4$, $g(n) = (100n)^3$
2. $f(n) = n^{1.01}$, $g(n) = n^{0.99} \cdot (\log n)^2$
3. $f(n) = 4n \cdot 2^n + n^{100}$, $g(n) = 3^n$
4. $f(n) = n^2 \cdot \log(n^2)$, $g(n) = n \cdot (\log n)^3$
5. $f(n) = 3^{n-1}$, $g(n) = 3^n$
6. $f(n) = 1.01^n$, $g(n) = n^2$
7. $f(n) = 2^{\log\log n}$, $g(n) = n$
8. $f(n) = (\log n)^{100}$, $g(n) = n^{0.001}$
9. $f(n) = 5n + \sqrt{n}$, $g(n) = 3n + \log n$
10. $f(n) = 2^n + \log n$, $g(n) = 2^n + (\log n)^{10}$
11. $f(n) = \sqrt[5]{n}$, $g(n) = \sqrt[3]{n}$
12. $f(n) = n!$, $g(n) = 3^n$
13. $f(n) = \log(15n!)$, $g(n) = n\log(n^9)$
14. $f(n) = \sum_{k=1}^{n} k$, $g(n) = \log(n!)$
15. $f(n) = \sum_{k=1}^{n} k^3$, $g(n) = n^3 \cdot \log n$

Solution:

Note 1: we introduce a new definition, small ω. We define $f = \omega(g)$ if and only if $g = o(f)$. Equivalently, $f = \omega(g)$ if and only if $\lim_{n\to\infty} f(n)/g(n) = \infty$ according to the definition of small-o.

Note 2: calculating the limit of $f(n)/g(n)$ is usually an efficient approach to determine their asymptotic relationship. Specifically, $\lim_{n\to\infty} f(n)/g(n) = 0$ implies that $f = o(g)$ and consequently $f = O(g)$. Symmetrically, $\lim_{n\to\infty} f(n)/g(n) = \infty$ implies that $f = \omega(f)$ and consequently $f = \Omega(g)$. Besides, $\lim_{n\to\infty} f(n)/g(n) = c > 0$ implies that $f = \Theta(g)$.

1. $f = \Omega(g)$: $\lim_{n\to\infty} \frac{n^4}{(100n)^3} = \lim_{n\to\infty} \frac{n}{100^3} = \infty$
2. $f = \Omega(g)$: $\lim_{n\to\infty} \frac{n^{1.01}}{n^{0.99}\cdot(\log n)^2} = \lim_{n\to\infty} \frac{n^{0.02}}{(\log n)^2} = \infty$
3. $f = O(g)$: $\lim_{n\to\infty} \frac{4n\cdot 2^n + n^{100}}{3^n} = \lim_{n\to\infty}(4n \cdot \frac{2}{3}^n + \frac{n^{100}}{3^n}) = 0 + 0 = 0$

4. $f = \Omega(g)$: $\lim_{n\to\infty} \frac{n^2 \cdot \log(n^2)}{n \cdot (\log n)^3} = \lim_{n\to\infty} \frac{2n}{(\log n)^2} = \infty$
5. $f = \Theta(g)$: $\lim_{n\to\infty} \frac{3^{n-1}}{3^n} = \lim_{n\to\infty} \frac{3^n}{3 \cdot 3^n} = \frac{1}{3}$
6. $f = \Omega(g)$: an exponential function always dominates a polynomial function.
7. $f = O(g)$: $\lim_{n\to\infty} \frac{2^{\log\log n}}{n} = \lim_{n\to\infty} \frac{2^{\log\log n}}{2^{\log n}} = 0$
8. $f = O(g)$: a fractional power function always dominates a polylogarithmic function.
9. $f = \Theta(g)$: $\lim_{n\to\infty} \frac{5n+\sqrt{n}}{3n+\log n} = \lim_{n\to\infty} \frac{5n}{3n} = \frac{5}{3}$
10. $f = \Theta(g)$: $\lim_{n\to\infty} \frac{2^n+\log n}{2^n+(\log n)^{10}} = \lim_{n\to\infty} \frac{2^n}{2^n} = 1$
11. $f = O(g)$: $\lim_{n\to\infty} \frac{\sqrt[5]{n}}{\sqrt[3]{n}} = \lim_{n\to\infty} \frac{1}{n^{\frac{2}{15}}} = 0$
12. $f = \Omega(g)$: $\lim_{n\to\infty} \frac{n!}{3^n} = \lim_{n\to\infty} \frac{1\times 2\times \cdots n}{3\times 3\times \cdots \times 3} = \infty$
13. $f = \Theta(g)$: $f = \log(15n!) = \Theta(\log(n!))$. Observe that $n! = 1 \times 2 \times 3 \cdot \cdots \cdot n \leq n \cdot n \cdot n \cdot \cdots \cdot n \leq n^n$ and assuming n is even (without loss of generality) $n! = 1 \times 2 \times 3 \cdot \cdots \cdot n \geq n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n - n/2) \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$. Hence $\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$. Then, $\frac{n}{2}\log\left(\frac{n}{2}\right) \leq \log(n!) \leq n\log n$ and $f = \Theta(\log(n!)) = \Theta(n\log n)$. $g = n\log(n^9) = 9n\log n = \Theta(n\log n)$. So, $f = \Theta(g)$.
14. $f = \Omega(g)$: First, $g = \Theta(n\log n)$ as shown above. So, $\lim_{n\to\infty} \frac{\sum_{k=1}^{n} k}{\log(n!)} = \lim_{n\to\infty} \frac{\frac{n(n+1)}{2}}{n\log n} = \lim_{n\to\infty} \frac{n}{\log n} = \infty$
15. $f = \Omega(g)$: $f = \frac{n^2(n+1)^2}{4} = \Theta(n^4)$, based on Faulhaber's formula. So, $f(n)$ dominates $g(n) = n^3 \cdot \log n$.

3. (16 pts.) Assume you have functions f, g and h. For each of the following statements, decide if you think it is true or false and give a proof or counterexample.

1. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
2. If $f(n) = \Theta(g(n))$, then $2^{f(n)} = \Theta(2^{g(n)})$.
3. If $f(n) = o(g(n))$, then $\log f(n) = o(\log g(n))$
4. If $f(n) = O(g(n))$, then $\frac{1}{f(n)} = \Omega(\frac{1}{g(n)})$

Solution:

1. True.
As $f(n) = O(g(n))$, there exist positive constants c_1 and N_1 such that $f(n) \leq c_1.g(n)$ for all $n \geq N_1$. Similarly, there exist positive constants c_2 and N_2 such that $g(n) \leq c_2.h(n)$ for all $n \geq N_2$. So for all n that $n \geq N_1$ and $n \geq N_2$, we have $f(n) \leq c_1.c_2.h(n)$. Replacing $c_1.c_2$ with c', we have $f(n) \leq c'.h(n)$. Therefore, $f(n) = O(h(n))$.
2. False.
Let's consider the counterexample, f(n) = 2n and g(n) = n. f (n) is $\Theta(g(n))$ in this case because we can find constants c=2 and N =1 such that $f(n) = cg(n)$ for all $n \geq N$. However, since $2^{f(n)} = 2^{2n}$ and $2^{g(n)} = 2^n$, $\lim \frac{2^{2n}}{2^n} = \infty$. So, $2^{f(n)}$ grows faster than $2^{g(n)}$ asymptotically.Thus the statement is false.
3. False.
A counterexample is f(n) = n and g(n) = n^2. f (n) is o(g(n)) in this case because $\lim_{n->\infty} \frac{n}{n^2} = 0$. However, $\log f(n) = \log n$ and $\log g(n) = \log n^2 = 2\log n$. As a result, $\lim_{n->\infty} \frac{\log n}{2\log n} = 1/2 \not\equiv 0$. So, the statement is false.

4. True.
As $f(n) = O(g(n))$, there exist positive constants c and N such that $f(n) \leq c.g(n)$ for all $n \geq N$. Rearranging the inequality gives, $\frac{1}{f(n)} \geq \frac{1}{cg(n)}$ => $\frac{1}{f(n)} \geq \frac{1}{c}(\frac{1}{g(n)})$. Replacing $\frac{1}{c}$ with c', we get $\frac{1}{f(n)} \geq c' \frac{1}{g(n)}$. Thus, $\frac{1}{f(n)} = \Omega(\frac{1}{g(n)})$.

4. (16 pts.) For each pseudo-code below, give the asymptotic running time in Θ notation. You may assume that standard arithmetic operations take $\Theta(1)$ time.

1.
```
for $i := 1$ to $n$ do
    $j := i$;
    while $j \leq n$ do
        $j := j + i$;
    end
end
```

2.
```
$i := 1$ ;
while $i \leq n$ do
    $j := 1$;
    while $j \leq i$ do
        $j := j + 1$;
    end
    $i := 2i$;
end
```

3.
```
$s := 0$;
for $i := 1$ to $n$ do
    for $j := i + 1$ to $n$ do
        for $k := j + 1$ to $n$ do
            $s := s + 1$;
        end
    end
end
```

4.
```
for $i := 1$ to $n^2$ do
    if $i$ mod $n = 0$ then
        $j := 1$;
        while $j \leq \frac{i}{n}$ do
            $j := j + 1$;
        end
    end
end
```

Solution:

Note: in general, to analyze the running time of nested loops you will need to represent the running time as a summation, calculate it as a closed form, and eventually simplify it using the asymptotic notations.

Specifically, say the outer loop goes with "for $i = 1$ to n", then we want to represent the running time of inner loop as a function of i, say $F(i)$, then the entire running time will be $\sum_{i=1}^{n} F(i)$.

1. For each i, j iterates from i to n stepping by i, i.e., i, $2i$, $3i$, ..., until it reaches n. Thus, there are $\lfloor \frac{n}{i} \rfloor$ iterations for each i. Thus, the running time is

$$\sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor \approx \sum_{i=1}^{n} \frac{n}{i} = \Theta(n \log n) \tag{1}$$

2. Assume $2^k \leq n < 2^{k+1}$ for some k. Then i iterates from 1 to 2^k multiplying by 2. i.e., 1, 2, 4, 8, ..., 2^k. For each i, j iterates from 1 to i. So, the running time is

$$\sum_{l=1}^{k} \sum_{j=1}^{2^l} 1 = \sum_{l=1}^{k} 2^l = \Theta(2^k) = \Theta(n) \tag{2}$$

   Here, l is the exponent of i, namely, $i = 2^l$.

3. There is one arithmetic operation for each (i, j, k) so the running time is

$$\begin{aligned} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} 1 &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} (n - j) && (3) \\ &= \sum_{i=1}^{n} (\sum_{j=i+1}^{n} n - \sum_{j=i+1}^{n} j) && (4) \\ &= \sum_{i=1}^{n} n(n-i) - \frac{1}{2}(n^2 + n - i^2 - i) && (5) \\ &= \sum_{i=1}^{n} \frac{n^2}{2} - (\frac{1}{2} + i)n + \frac{i^2 + i}{2} && (6) \\ &= \frac{n^3}{2} - \frac{n^2(n+2)}{2} + \frac{n(n+1)(n+2)}{6} && (7) \\ &= \frac{1}{6} n(n^2 - 3n + 2) = \Theta(n^3) && (8) \end{aligned}$$

4. j iterates only if i is a multiple of n. Let k be a number such that $i = kn$, that is, $k = \frac{i}{n}$. Then, j iterates from 1 to k. The running time is

$$\sum_{i=1}^{n^2} 1 + \sum_{k=1}^{n} k = n^2 + \frac{n(n+1)}{2} = \Theta(n^2) \tag{9}$$

Rubrics:

Problem 2, 15 pts

Each part has 1 point.
1 - Provide a correct answer

Problem 3, 16 pts

Each part has 4 points.
2 - Provide an appropriate proof or counterexample
2 - Provide a correct answer

Problem 4, 16 pts

Each part has 4 points.
2 - Provide an appropriate proof or justification
2 - Provide a correct answer

CMPSC 465 Data Structures & Algorithms
Spring 2022 Chunhao Wang and Mingfu Shao

Solution 02

1. (0 pts.) Acknowledgements. The assignment will receive a 0 if this question is not answered.

1. If you worked in a group, list the members of the group. Otherwise, write "I did not work in a group."
2. If you received significant ideas about your solutions from anyone not in your group, list their names here. Otherwise, write "I did not consult anyone except my group members".
3. List any resources besides the course material that you consulted in order to solve the material. If you did not consult anything, write "I did not consult any non-class materials."

2. (10 pts.) Solve each of the following recursions using master's theorem. Give the closed form of $T(n)$ in Big-Theta notation; you don't need to show your middle steps.

1. $T(n) = 8 \cdot T(n/2) + 100 \cdot n^3$.
2. $T(n) = 8 \cdot T(n/2) + 1000 \cdot n^{3.5}$.
3. $T(n) = 16 \cdot T(n/2) + n \cdot \log(n)$
4. $T(n) = 2 \cdot T(n/2) + n \cdot \log(n)$.
5. $T(n) = 8 \cdot T(n/2) + n^{3.5} \cdot \log^2(n)$.

Solution:

The first 2 subproblems follow recursion $T(n) = a \cdot T(n/b) + \Theta(n^d)$, and the last 3 subproblems follow recursion $T(n) = a \cdot T(n/b) + \Theta(n^d \cdot \log^s n)$.

1. $a = 8, b = 2, d = 3$, $\log_b a = \log_2 8 = 3$, $d = \log_b a$, $T(n) = \Theta(n^d \log n) = \Theta(n^3 \log n)$.
2. $a = 8, b = 2, d = 3.5$, $\log_b a = \log_2 8 = 3$, $d > \log_b a$, $T(n) = \Theta(n^d) = \Theta(n^{3.5})$.
3. $a = 16, b = 2, d = 1, s = 1$, $\log_b a = \log_2 16 = 4$, $d < \log_b a$, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^4)$.
4. $a = 2, b = 2, d = 1, s = 1$, $\log_b a = \log_2 2 = 1$, $d = \log_b a$, $T(n) = \Theta(n^d \log^{s+1} n) = \Theta(n \log^2 n)$.
5. $a = 8, b = 2, d = 3.5, s = 2$, $\log_b a = \log_2 8 = 3$, $d > \log_b a$, $T(n) = \Theta(n^d \log^s n) = \Theta(n^{3.5} \log^2 n)$.

3. (10 pts.) Suppose you have m sorted arrays, each with n elements, and you want to combine them into a single sorted array with mn elements.

1. If you do this by merging the first two arrays, next with the third, then with the fourth, until in the end with the last one. What is the time complexity of this algorithm, in terms of m and n? Please also provide the analysis.
2. Give a more efficient solution to this problem, using divide-and-conquer. What is the running time? Please also provide the analysis.

Solution:

1. Each merge step requires $O(x+y)$ time for x elements in first array and y elements in the second array.

   Extending this, we can see that initially we start with $2n$ being the time required to merge the first two arrays. Then to merge with the third array it would take $3n$, and then $4n$ for the fourth and so on. Thus, it can be represented as:

$$T(m,n) = 2n+3n+\cdots+(m-1)n+mn = n(2+3+\cdots+(m-1)+m) = n\left(\frac{m(m+1)}{2}-1\right) = \Theta(nm^2)$$

2. Without loss of generality, recursively divide all the arrays into two sets, each with $m/2$ arrays. Then, recursively combine the arrays within the two sets and finally merge the resulting two sorted arrays into the output array. The base case of the recursion is $m=1$, where it's just a single array and no merging needs to take place. This procedure is illustrated in the pseudocode below (though not required from the answer).

```
Algorithm 1: mergeArrays($A$, $num$)
Input  : An array $A$ containing $num$ sorted arrays, each with $n$ elements
Output: A single sorted array with all the elements in all the arrays in $A$
if $num$ = 1 then
    return $A[0]$
else
    $merged1 \leftarrow$ mergeArrays ($[A[0], \cdots, A[\lceil \frac{num}{2} \rceil - 1]]$, $\lceil \frac{num}{2} \rceil$)
    $merged2 \leftarrow$ mergeArrays ($[A[\lceil \frac{num}{2} \rceil], \cdots, A[num-1]]$, $\lfloor \frac{num}{2} \rfloor$)
    return merge-two-sorted-arrays($merged1$, $merged2$)
end
```

   Note: From this point, we use $T(m)$ to represent the running time. Although n is an input of our problem, the array size is constant across all of them. Also, with only one variable in the running time expression, it's more similar to the cases we see in the class where we apply the Master's theorem.

   For merging the two arrays, i.e., $merged1$ and $merged2$, we can see that it will be $\theta(nm)$. Thus, the running time is given by $T(m) = 2T(m/2) + \theta(nm)$. By the Master theorem, $T(m) = \Theta(nm \log m)$.

4. (10 pts.) If we add one more partition step in find-pivot function in selecting problem, and use 3 as the size of each subarray, the algorithm would become:

```
function find-pivot ($A$, $k$)
    Partition $A$ into $n/3$ subarrays;
    Let $M$ be the list of medians of these $n/3$ subarrays;
    Partition $M$ into $n/9$ subarrays;
    Let $M'$ be the list of medians of these $n/9$ subarrays;
    return selection($M'$, $|M'|/2$)
end function;
```

Analyze the running time of the new selection algorithm with the find-pivot function described above.

Solution. Let m be the median of array M'. Consider the numbers in M. In half of the $n/9$ subarrays, i.e., those with median that is less than m, there are two numbers (i.e., the median of this subarray and another number) that are guaranteed less than m. This amounts to $2 \cdot (n/9)/2 = n/9$ elements in M that are less than m. Now consider the numbers in A. Recall that each element in M is a median of a subarray of size 3 in A. As we already show that there are $n/9$ elements in M that are less than m, in the corresponding subarrays, there are 2 numbers (the median and another number) that are guaranteed less than m. This amounts to $2 \cdot n/9 = 2n/9$ elements in A that are less than or equal to m. Thus, there are at most $7n/9$ elements are larger than m in A. Symmetrically, there are at most $7n/9$ elements are smaller than m in A.

The recursive call of selection function in above algorithm takes $T(n/9)$ time as $|M'| = n/9$. The recurrence relation for the entire algorithm is: $T(n) \leq \Theta(n) + T(n/9) + T(7n/9)$, which gives $T(n) = \Theta(n)$.

5. (10 pts.) Suppose you are given an array $A[1 \cdots n]$ of integers which can be positive, negative or 0. A sub-array is a contiguous sequence of elements from A. In particular, for any two indexes i and j with $1 \leq i \leq j \leq n$, $A[i \cdots j]$ is the sub-array that starts at index i and ends at j. The sum of the sub-array $A[i \cdots j]$ is the sum of all the numbers it contains: $A[i] + A[i+1] + ... + A[j]$. For example, if $A = [5, 1, -3, -2, 4, 0]$, the sum of $A[0 \cdots 3]$ is 1 and the sum of $A[1 \cdots 4]$ is 0. Given A, design a divide-and-conquer algorithm to find the sub-array of minimum sum. For example, in the array $A = [5, 1, -3, -2, 4, 0]$, the sub-array of minimum sum is $A[2, 3]$ with sum -5. Your algorithm should run in $O(n \log n)$ time.

Solution. We can solve this problem by following merge sort algorithm with few changes. Let's divide the array into 2 halves, splitting at middle element(say 'mid'). Now the minimum contiguous sub-array has 3 possible scenarios.

- Present in the Left half
- Present in the Left half
- sub-array starts with an element in the left half and ending at some element in the right half.

The first 2 cases can be recursively solved to find the subproblem result. For the third case, we can find the minimum crossing sum by finding the minimum contiguous sum subarray starting at some point in left half and ending at mid, another starting at mid + 1 and ending at some point in the right half, and combine both to return the crossing sum. Now our solution is the minimum value from all the three cases.

```
Algorithm 2: minSumSubArray(A,left,right)
Input  : An array A[] of n integers
Output: minimum sum of the contiguous subarray in $A[left \cdots right]$
if left = right then
    return $A[left]$
end
$mid \leftarrow (left + right)/2$;
$l \leftarrow minSumSubArray(A, left, mid)$;
$r \leftarrow minSumSubArray(A, mid + 1, right)$;
$c \leftarrow crossingSum(A, left, mid, right)$;
return $min(l, r, c)$;
```

At each iteration the problem size gets reduced to half and finding the crossing sum requires linear time as we would be visiting each element at most once. So the recurence relation becomes, $T(n) = 2T(n/2) + \Theta(n)$. Following the master theorem we can see that the time complexity is $O(n \log n)$.

Algorithm 3: crossingSum (A,left,mid,right)

Input : An array A and three positions left, mid, right

Output: minimum sum of the contiguous subarray in $A[left \cdots right]$ that span the mid position

```
$i \leftarrow mid - 1$;
$leftSum \leftarrow A[mid]$;
$leftSumTemp \leftarrow A[mid]$;
while $i \geq left$ do
    if $leftSumTemp + A[i] < leftSum$ then
        $leftSum \leftarrow leftSumTemp + A[i]$;
    end
    $leftSumTemp \leftarrow leftSumTemp + A[i]$;
    $i \leftarrow i - 1$;
end
$j \leftarrow mid + 2$;
$rightSum \leftarrow A[mid + 1]$;
$rightSumTemp \leftarrow A[mid + 1]$;
while $j \leq right$ do
    if $rightSumTemp + A[j] < rightSum$ then
        $rightSum \leftarrow rightSumTemp + A[j]$;
    end
    $rightSumTemp \leftarrow rightSumTemp + A[j]$;
    $j \leftarrow j + 1$;
end
return $leftSum + rightSum$;
```

6. (0 pts.) *(NOTE: you don't need to submit your solution for this problem.)* Consider recurrence relation $T(n) = \Theta(n) + T(a \cdot n) + T(b \cdot n)$, $T(1) = 1$, $0 < a < 1$, $0 < b < 1$. Prove the following:

1. $T(n) = \Theta(n)$, if $a + b < 1$.
2. $T(n) = \Theta(n \cdot \log n)$, if $a + b = 1$.

Solution. Assume that the term $\Theta(n)$ in the recursion admits a lower bound of $d_1 n$ and upper bound of $d_2 n$, for large enough n. That is $T(n) \geq d_1 n + T(an) + T(bn)$ and $T(n) \leq d_2 n + T(an) + T(bn)$.

1. We first prove that $T(n) = O(n)$, by induction. Assume that $T(n) \leq c_2 n$ for large enough n, where $c_2 = d_2/(1 - a - b)$. Now we prove that $T(n+1) \leq c_2(n+1)$. We can write

$$T(n+1) \leq d_2(n+1) + T(a(n+1)) + T(b(n+1)).$$

We have $a(n+1) \leq n$ and $b(n+1) \leq n$ for large enough n as $a < 1$ and $b < 1$. By the inductive assumption we have

$$\begin{aligned} T(n+1) &\leq d_2(n+1) + c_2 a(n+1) + c_2 b(n+1) \\ &\leq d_2(n+1) + c_2(a+b)(n+1) \\ &= (d_2 + c_2(a+b))(n+1) \\ &= c_2(n+1). \end{aligned}$$

It's easy to verify the last equation, i.e., $d_2 + c_2(a+b) = c_2$ with $c_2 = d_2/(1-a-b)$. In fact this is where we determine the value of c_2. You can also see why $a + b < 1$ is required here. We then prove that $T(n) = \Omega(n)$, by induction. Assume that $T(n) \geq c_1 n$ for large enough n, where $c_1 = d_1/(1-a-b)$. Now we prove that $T(n+1) \geq c_1(n+1)$. We can write

$$\begin{aligned} T(n+1) &\geq d_1(n+1) + T(a(n+1)) + T(b(n+1)) \\ &\geq d_1(n+1) + c_1 a(n+1) + c_1 b(n+1) \\ &\geq d_1(n+1) + c_1(a+b)(n+1) \\ &= (d_1 + c_1(a+b))(n+1) \\ &= c_1(n+1). \end{aligned}$$

The last equation holds as $d_1 + c_1(a+b) = c_1$ with $c_1 = d_1/(1-a-b)$.

2. We first prove that $T(n) = O(n \log n)$, by induction. Assume that $T(n) \leq c_2 n \log n$ for large enough n, where $c_2 = -d_2/(a \log a + b \log b)$. Now we prove that $T(n+1) \leq c_2(n+1)\log(n+1)$. We can write

$$\begin{aligned} T(n+1) &\leq d_2(n+1) + T(a(n+1)) + T(b(n+1)) \\ &\leq d_2(n+1) + c_2 a(n+1)\log(a(n+1)) + c_2 b(n+1)\log(b(n+1)) \\ &\leq d_2(n+1) + c_2(a \log a + b \log b)(n+1) + c_2(a+b)(n+1)\log(n+1) \\ &= (d_2 + c_2(a \log a + b \log b))(n+1) + c_2(n+1)\log(n+1) \\ &= c_2(n+1)\log(n+1). \end{aligned}$$

The last equation holds as $d_2 + c_2(a \log a + b \log b) = 0$ with $c_2 = -d_2/(a \log a + b \log b)$. We then prove that $T(n) = \Omega(n \log n)$, by induction. Assume that $T(n) \geq c_1 n \log n$ for large enough n, where

$c_1 = -d_1/(a \log a + b \log b)$. Now we prove that $T(n+1) \geq c_1(n+1)\log(n+1)$. We can write

$$\begin{aligned}
T(n+1) &\geq d_1(n+1) + T(a(n+1)) + T(b(n+1)) \\
&\geq d_1(n+1) + c_1 a(n+1)\log(a(n+1)) + c_1 b(n+1)\log(b(n+1)) \\
&\geq d_1(n+1) + c_1(a\log a + b\log b)(n+1) + c_1(a+b)(n+1)\log(n+1) \\
&= (d_1 + c_1(a\log a + b\log b))(n+1) + c_1(n+1)\log(n+1) \\
&= c_1(n+1)\log(n+1).
\end{aligned}$$

The last equation holds as $d_1 + c_1(a\log a + b\log b) = 0$ with $c_1 = -d_1/(a\log a + b\log b)$.

Rubrics:

Problem 2, 10 pts

Each part has 2 points.
2 - Provide a correct answer

Problem 3, 10 pts

1. 2 points: Finds the correct running time to merge two arrays.
   2 points: Provides the correct final answer.
2. 2 points: Identifies the size and the number of sub-problems that can result in a more efficient algorithm.
   2 points: Provides the correct merging time for each recursion.
   2 points: Provides the correct running time for the improved algorithm.

Problem 4, 10 pts

1. 5 points : Analyzed the function find-pivot appropriately.
2. 3 points : Provided a proper recurrence.
3. 2 points : Provided the answer.

Problem 5, 10 pts

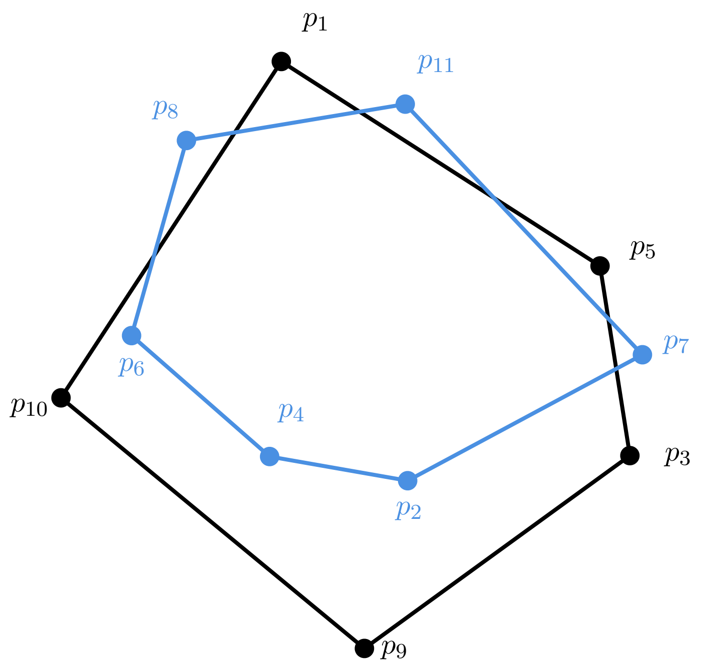
1. 3 points : identified 3 possible cases.
2. 3 points : Analysed/explained crossing sum(case 3)
3. 2 points : Provided proper recurrence.
4. 2 points : solved the recurrence for time complexity.
5. 4 points : correct runtime analysis without master theorem. theorem/recurrence

Students can provide algorithm/pseudocode, but they should explain the correctness or how the algorithm works. The 5th case in rubric for problem 5 is when students follow any other methods(tree analysis, iterative analysis) to analyse the running time and get the answer correctly.

Solution 03

1. (15 pts.) Run the combine step of the divide-and-conquer algorithm for convex hull on the instance given below. You are given $C_1 = (p_1, p_{10}, p_9, p_3, p_5)$ and $C_2 = (p_8, p_6, p_4, p_2, p_7, p_{11})$.

1. Find the lowest point p^* in $C_1 \cup C_2$.
2. Transform C_1 into C_1' so that points in C_1' is sorted in increasing angle w.r.t. p^*.
3. Partition C_2 into two lists C_{2a} and C_{2b} so that each list is sorted in increasing angle w.r.t. p^*.
4. Give list C_2' by merging C_{2a} and C_{2b} so that each points in C_2' is sorted in increasing angle w.r.t. p^*.
5. Give list C' by merging C_1' and C_2' so that each points in C' is sorted in increasing angle w.r.t. p^*.
6. Run Graham-Scan-Core algorithm to find convex hull of C'. Show stack operations at each step (to deal with each point). For example, you need to write like "For A: push A; pop B", which indicates when you process point A, push A into stack and also pop B out.



Solution:

1. p_9 is the point with lowest y-coordinate.
2. $C_1' = (p_9, p_3, p_5, p_1, p_{10})$
3. $C_{2a} = (p_7, p_{11}, p_8)$, $C_{2b} = (p_2, p_4, p_6)$. You may put p_7 and p_6 in either C_{2a} or C_{2b}.
4. $C_2' = (p_7, p_2, p_{11}, p_8, p_4, p_6)$
5. $C' = (p_9, p_3, p_7, p_5, p_2, p_1 1, p_1, p_8, p_4, p_6, p_{10})$.
6. Elements in the stack are listed from stack bottom to top.
   For p_9, push p_9, stack: p_9;
   For p_3, push p_3, stack: p_9, p_3;
   For p_7, push p_7, stack: p_9, p_3, p_7;
   For p_5, push p_5, stack: p_9, p_3, p_7, p_5;
   For p_2, push p_2, stack: p_9, p_3, p_7, p_5, p_2;
   For p_{11}, pop p_2, push p_{11}, stack: $p_9, p_3, p_7, p_5, p_{11}$;
   For p_1, push p_1, stack: $p_9, p_3, p_7, p_5, p_{11}, p_1$;
   For p_8, push p_8, stack: $p_9, p_3, p_7, p_5, p_{11}, p_1, p_8$;
   For p_4, push p_4, stack: $p_9, p_3, p_7, p_5, p_{11}, p_1, p_8, p_4$;
   For p_6, pop p_4, push p_6, stack: $p_9, p_3, p_7, p_5, p_{11}, p_1, p_8, p_6$;
   For p_{10}, pop p_6, push p_{10}, stack: $p_9, p_3, p_7, p_5, p_{11}, p_1, p_8, p_{10}$;

2. (10 pts.) Given n points $p_1, p_2, ..., p_n$ in 2D-plane and a slope a, we want to find two *closest* lines with functions $y = ax + b_1$ and $y = ax + b_2$, $b_1 \leq b_2$, so that all n points are in or on between these two lines. Construct an algorithm with the time complexity $O(n)$ and justify your algorithm. Describe your algorithm (you do not need to give a pseudo code) and analyze the running time of your algorithm.

Solution: For given n points, let $p_i = (p_{ix}, p_{iy})$ and let $P^* = \{p_i^* : i = 1, 2, ..., n\}$ be a set of duals of p_i. Then, consider the upper- and lower-envelop of P^*, denoted as $UE(P^*)$ and $LE(P^*)$, respectively.
We now are interested in lines with the fixed slope a. Since a line $y = ax + b$ corresponds to $(a, -b)$ in the dual plane, the desired lines will be on the line $x = a$ in the dual plane. In other words, if we let $l_1 : y = ax + b_1$ and $l_2 : y = ax + b_2$, then $l_1^* = (a, -b_1), l_2^* = (a, -b_2)$ are on the vertical line $x = a$ in the dual plane.
If some point l^* lies on $UE(P^*)$, it means that l^* is above all lines p_i^* in P^*. Thus, by the propery of the duality, all points p_i is above the line l. Same goes true for the lower-envelope. Therefore, if we find two points on the line $x = a$ in the dual plane that one is on the upper-envelop and the other one is on the lower-envelop, their duals have a slope a and all points p_i are contained between them. As we need to find two cloesest lines, therefore, l_1^* = the intersection of $UE(P^*)$ and $x = a$, and l_2^* = the intersection of $LE(P^*)$ and $x = a$ in the dual plane. In other words, $l_1^* = (a, \max_{1 \leq i \leq n}(p_{ix}a - p_{iy}))$ and $l_2^* = (a, \min_{1 \leq i \leq n}(p_{ix}a - p_{iy}))$. Transforming to the original plane gives $b_1 = -\max_{1 \leq i \leq n}(p_{ix}a - p_{iy}))$ and $b_2 = -\min_{1 \leq i \leq n}(p_{ix}a - p_{iy})$.
From this argument, it suffices to construct an algorithm to find the maximum and minimum of $p_i.xa - p_i.y$. It takes $O(n)$ as we need to iterate once and get the maximum and minimum value of them.

3. (10 pts.)

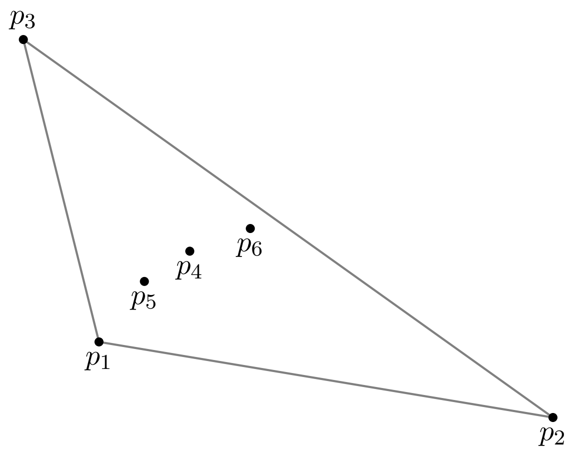
1. Let P be a set of n points $(p_1, p_2, \cdots, p_n)$ in a plane. A point $p_i \in P$ is *maximal* if no point in P is both above and to the right of p_i. Give a set P with 6 points such that there is only one point on the convex hull of P but this point is not maximal, and there is only one point that is maximal but not on

the convex hull of P. Include a visual illustration of these 6 points and the convex hull. Specifically, point out which is the point that is on the convex hull but not maximal, and which is the maximal point but not on the convex hull.

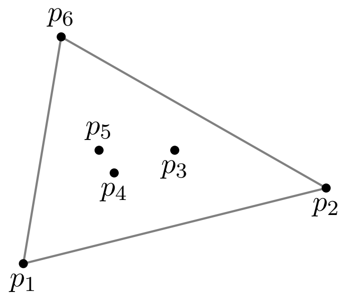
2. Design an instance of convex hull problem with 6 points, such that if Graham-Scan algorithm runs on your instance, the sequence of stack operations is (push, push, push, push, pop, push, pop, pop, push). Include a visual illustration of these 6 points and the convex hull.

Solution: *Both of these parts can have multiple viable solutions*. Below are simply example answers for these two parts, respectively.

1. p_1 is the point that is on the convex hull but not maximal. p_6 is the maximal point that is not on the convex hull.
   Thinking process: As we know from class, we need at least 3 points to form a convex hull. So, to find these 6 points, we start with the simplest setting i.e. to find 4 points with one maximal but non-convex point and one convex but non-maximal point. Then, we add 2 other points that don't break the requirements we fulfilled with those 4 points earlier.



2. Thinking process: Based on the Gramham-Scan-Core pseudo-code covered in class and the stack operation sequence, we can deduct the positional relationship between these 6 points.



- Since the 4th operation is push, $p_2 \to p_3 \to p_4$ is "turning left".
- Since the 5th operation is pop, $p_3 \to p_4 \to p_5$ is "turning right".
- Since the 6th operation is push, $p_2 \to p_3 \to p_5$ is "turning left".
- Since the 7th operation is pop, $p_3 \to p_5 \to p_6$ is "turning right".
- Since the 8th operation is pop, $p_2 \to p_3 \to p_6$ is "turning right".
- Since the 9th operation is push, $p_1 \to p_2 \to p_6$ is "turning left".

4. (10 pts.) You are given n points $P = \{p_1, p_2, \cdots, p_n\}$ on 2D plane, represented as their coordinates. You are informed that the convex hull of P contains $O(\sqrt{\log n})$ points in P. Design an algorithm to compute the convex hull of P in $O(n \cdot \sqrt{\log n})$ time. You may assume that no three points in P are on the same line.

Solution: We first find the point in P with smallest y-coordinate, denoted as p_1^*. Clearly, p_1^* must be on the convex hull of P. Let $C = (p_1^*)$ store the list of points on the convex hull of P found so far. Let $p_0^* = p_1^* - (1, 0)$ be a virtual point (not within P) be the point on the left of p_1^*. We then iteratively find all other points on the convex hull of P. In the k-th iteration, $k = 2, 3, \cdots$, for each point $p \in P \setminus \{p_0^*, p_1^*\}$, we compute the angle $\angle pp_{k-1}^*p_{k-2}^*$. We then find the point that maximizes this angle, denoted as p_k^*. Clearly, p_k^* must be on the convex hull of p as well (*Proof.* line $p_k^*p_{k-1}^*$ is an indicator: all other points locate at one side of this line). Therefore, we add p_k^* to C and continue to next iteration. The algorithm terminates when we find p_k^* equals to p_1^*.

Each iteration finds a new point on the convex hull and takes linear time. The total number of iterations equals to the number of points on the convex hull. Hence, the running time is $O(n \cdot s)$, where s is the number of points on the convex hull. This algorithm is *output-sensitive*. When there are $O(\sqrt{\log n})$ number of points on the convex hull, as described in this problem, this algorithm runs in $O(n \cdot \sqrt{\log n})$ time.

5. (0 pts.) *(NOTE: you don't need to submit your solution for this problem.)* Given two *sorted* arrays A and B of size m and n respectively, and an integer k, $1 \le k \le m + n$, design an algorithm to find the k-th smallest number in A and B. Describe your algorithm and analyze the running time of your algorithm. Your algorithm should run in $O(\log(m + n))$ time.

Solution: We cannot afford merging A and B, as it takes linear time rather than the desired logarithmic time. The idea of the algorithm is to reduce k by half using a single comparison. Specifically, each time we compare $A[mid_1 - 1]$ and $B[mid_2 - 1]$, where $mid_1 = \min(m, k/2)$ and $mid_2 = \min(n, k/2)$. If $A[mid_1 - 1] > B[mid_2 - 1]$, we eliminate all the elements before $B[mid_2]$, because it is impossible for the k-th smallest value to be located before and including $B[mid_2 - 1]$; otherwise, we eliminate all the elements before $A[mid_1]$.

Define function find-kth-smallest(A, m, B, n, k) return the k-th smallest number of $A[0 \cdots m)$ and $B[0 \cdots n)$. We assume A and B start with index 0. We also assume A and B represent "pointers" to the array, i.e., we will use $A + 3$ to represents the array shifted 3 elements. The pseudocode is shown below:

```
function: find-kth-smallest($A, m, B, n, k$)
if $m == 0$ then
    return $B[k - 1]$
end
if $n == 0$ then
    return $A[k - 1]$
end
if $k == 1$ then
    return $\min(A[0], B[0])$
end
$mid_1 = \min(m, k/2)$, $mid_2 = \min(n, k/2)$;
if $A[mid_1 - 1] > B[mid_2 - 1]$ then
    return find-kth-smallest($A, m, B + mid_2, n - mid_2, k - mid_2$)
end
return find-kth-smallest($A + mid_1, m - mid_1, B, n, k - mid_1$)
```

In above algorithm, at each recursive level, either k is halved, or one array is eliminated. The running time is $T(k) = T(k/2) + \Theta(1)$, which gives $T(k) = \Theta(\log k)$.

Rubrics:

Problem 1, 15 pts

1. 1 point : identified p* correctly.
2. 2 points : C_1' is correct
3. 3 points : C_{2a} and C_{2b} are correct.
4. 2 points : C_2' is correct.
5. 2 points : C_1' is correct. theorem/recurrence
6. 5 points : stack operations at each step are correct
7. 3 points : At least 7 operations are correct.
8. 2 points : at least 5 operations are correct.
9. 1.5 points : I don't know how to answer this question.

Problem 2, 10 pts

1. 5 points : Provided a proper proof or justification of an algorithm (Although they don't use the duality to justify, they would get full scores if it's correct)
2. 3 points : Provided $O(n)$ algorithm
3. 2 points : Analyzed the running time correctly
4. 1 point : I don't know how to answer this question.

Problem 3, 10 pts

1. - 2 points: Provided a correct visual illustration of the 6 points.
   - 1 point: Drew the convex hull in the visual illustration.
   - 1 point: Correctly pointed out which point is on the convex hull but not maximal.
   - 1 point: Correctly pointed out which point is maximal but not on the convex hull.
   - 0.5 points : I don't know how to answer this question.
2. - 3 points: The 6 points have the correct positional relationships. (See "thinking process" in the solution for the positional relationships)
   - 2 points: Drew the convex hull in the visual illustration i.e. p_1, p_2, and p_6 are on the convex hull; the other points are inside the convex hull.
   - 0.5 points : I don't know how to answer this question.

Problem 4, 10 pts

1. 10 points: Provided $O(n\sqrt{logn})$ algorithm.
2. 4 points: Incorrect design but provided appropriate running time analysis (e.g. the running time is $O(n \cdot s)$, where s is the number of points on the convex hull).
3. 1 point : I don't know how to answer this question.

CMPSC 465 Data Structures & Algorithms
Spring 2022 Chunhao Wang and Mingfu Shao

Solution 04

0. (0 pts.) Acknowledgements. The assignment will receive a 0 if this question is not answered.

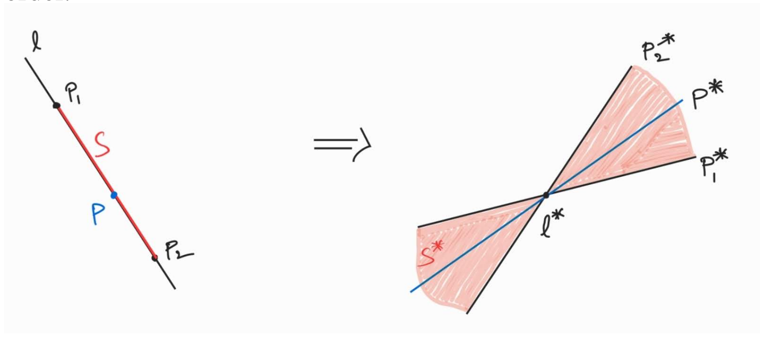
1. If you worked in a group, list the members of the group. Otherwise, write "I did not work in a group."
2. If you received significant ideas about your solutions from anyone not in your group, list their names here. Otherwise, write "I did not consult anyone except my group members".
3. List any resources besides the course material that you consulted in order to solve the material. If you did not consult anything, write "I did not consult any non-class materials."

1. (15 pts.) In class, we learned the following property about duality: point p is on line l if and only if point l^* is on line p^*.

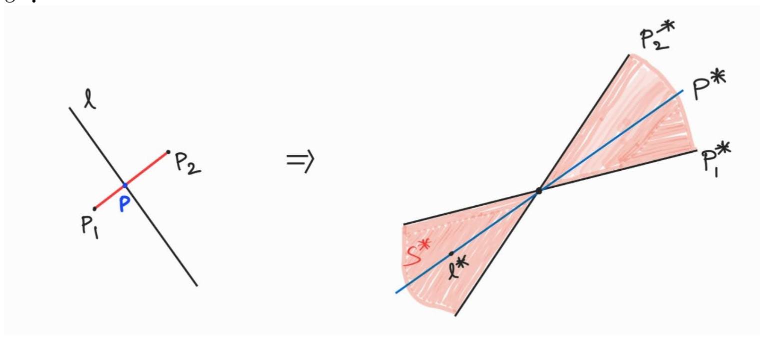
1. Using the property above, prove the following properties:
   (a) If n points $p_1, ..., p_n$ are on a common line l, then $p_1^*, p_2^*, ..., p_n^*$ intersect at a common point l^*.
   (b) If n lines $l_1, ..., l_n$ intersect at a common point p, then $l_1^*, l_2^*, ..., l_n^*$ are on a common line p^*.
2. If we have a line segment s connecting two points p_1 and p_2, describe a region s^* corresponding to the dual of s in terms of p_1^* and p_2^*. (No rigorous proof is needed.)
3. If a line l intersects the line segment s, prove that l^* is in the region s^*.

Solution:

1. (a) For each p_i, since p_i is on the line l, l^* must be on the line p_i^* in the dual plane. This must hold for every p_i, so l^* must be on the lines $p_1^*, p_2^*, ... ,p_n^*$. Therefore, $p_1^*, p_2^*, ..., p_n^*$ intersect at the common point l^*.
   (b) For each l_i, since p is on the line l_i, l_i^* must be on the line p^* in the dual plane. This must hold for every l_i, so l_1^*, l_2^*, ..., l_n^* must be on the line p^*. Therefore, $l_1^*, l_2^*, ..., l_n^*$ are on the common line p^*.
2. The line segment s coincides with a line connecting two points p_1 and p_2. Let us say this line l. Obviously, p_1^* and p_2^* intersects at the point l^* in the dual plain. To figure out the region s^*, we investigate where the points on the line segment s will be transformed to the dual plane. For every point p on s, p is also on the line l. Thus, p^* passes through the point l^*. Also, p is in between p_1 and p_2, so the slope of p^* is in between the ones of p_1^* and p_2^*. It means that p^* should be placed in the region between p_1^* and p_2^*. This must hold for every p on s, hence, s is transformed to the area between p_1^* and p_2^*. Without loss of generality, assume p_1 is located at the left to the p_2, that is, the x coordinate of p_1 is less than the one of p_2. Then, s^* is the region sweeping from p_1^* to p_2^* in counter-clockwise order.



3. Since the line l intersects the line segment s, there should be a point p on l and s. We observe that p^* is in the region s^* from above. Also, we know that l^* is on the line p^*, which means l^* is in the region s^*.



2. (15 pts.) We are given a graph $G = (V, E)$; G could be a directed graph or undirected graph. Let M be the adjacency matrix of G. Let n be the number of vertices so that the matrix M is $n \times n$ matrix. For any matrix A, let us denote the element of i-th row and j-th column of the matrix A by $A[i, j]$.

1. Consider the square of the adjacency matrix M. For all i and j, show that $M^2[i, j]$ is the number of different paths of length 2 from the i-th vertex to the j-th vertex. It should be explained or proved as clearly as possible.
   Solution: For any i and j, let us find the number of different paths of length 2 from i to j. Assume we go through k during our trip. Then, there should be edges between i and k and between k and j. In other words, $M[i, k] \cdot M[k, j] = 1$. This path $i \to k \to j$ is one of the paths from i and j. We need to consider all k for counting the total number of paths, so it should be

$$\sum_{k=1}^{n} M[i, k] \cdot M[k, j]$$

   It turns out that it is equal to $M^2[i, j]$.

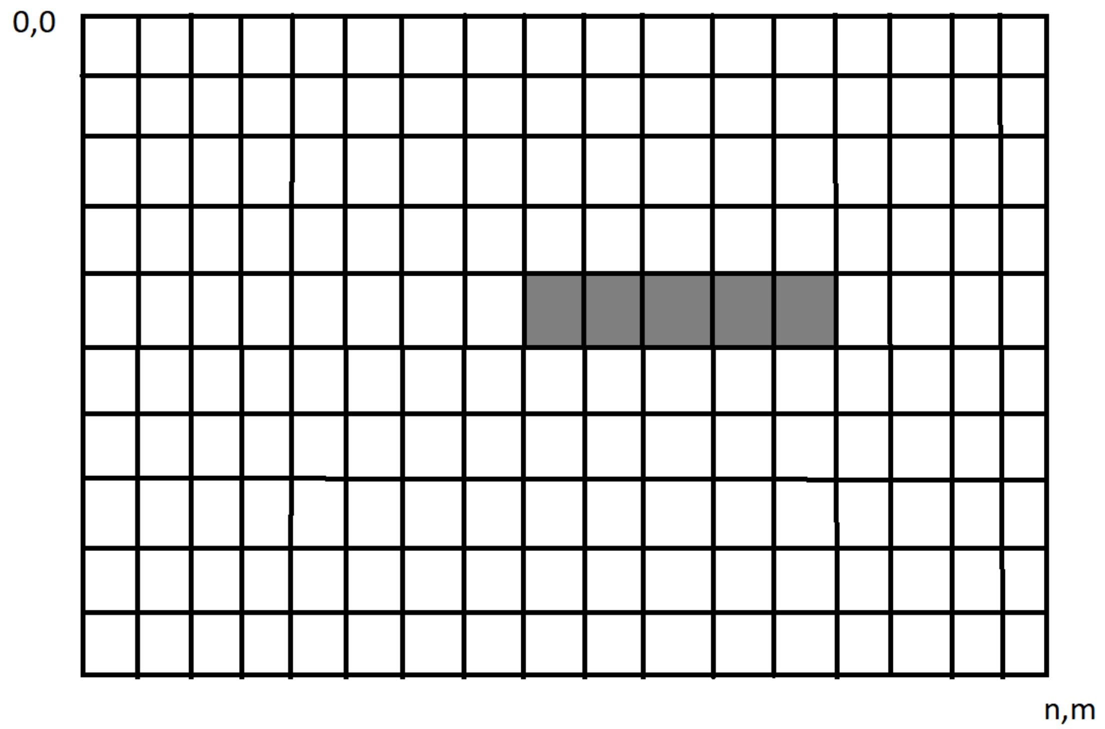
2. For any positive integer k, show that $M^k[i, j]$ is the number of different paths of length k from the i-th vertex to the j-th vertex. You may use induction on k to prove it.
   Solution: We will use induction on k.
   (a) (Base case) $k = 1$
   It is clear that $M^k[i, j] = M[i, j]$ has the number of different paths from i to j.
   (b) (Inductive step)
   Assume the result is true for $k = l$. It means that $M^l[i, j]$ is the number of different paths of length l from i to j. Note that $M^{l+1}[i, j] = \sum_{s=1}^{n} M^l[i, s] \cdot M[s, j]$ from the matrix multiplication $M^{l+1} = M^l \cdot M$. Every path of length $l + 1$ from i to j consists of paths from i to some vertex s of length l and an edge s to j. By the induction hypothesis, $M^l[i, s]$ has the number of different paths of length l from i to s. Thus, $M^l[i, s] \cdot M[s, j]$ has the number of different paths of length $l + 1$ that is from i to j and located at s at the l-th step. Since s is arbitrary, we need to add up over s so $\sum_{s=1}^{n} M^l[i, s] \cdot M[s, j]$ is the total number of different paths of length $l + 1$ from i to j.

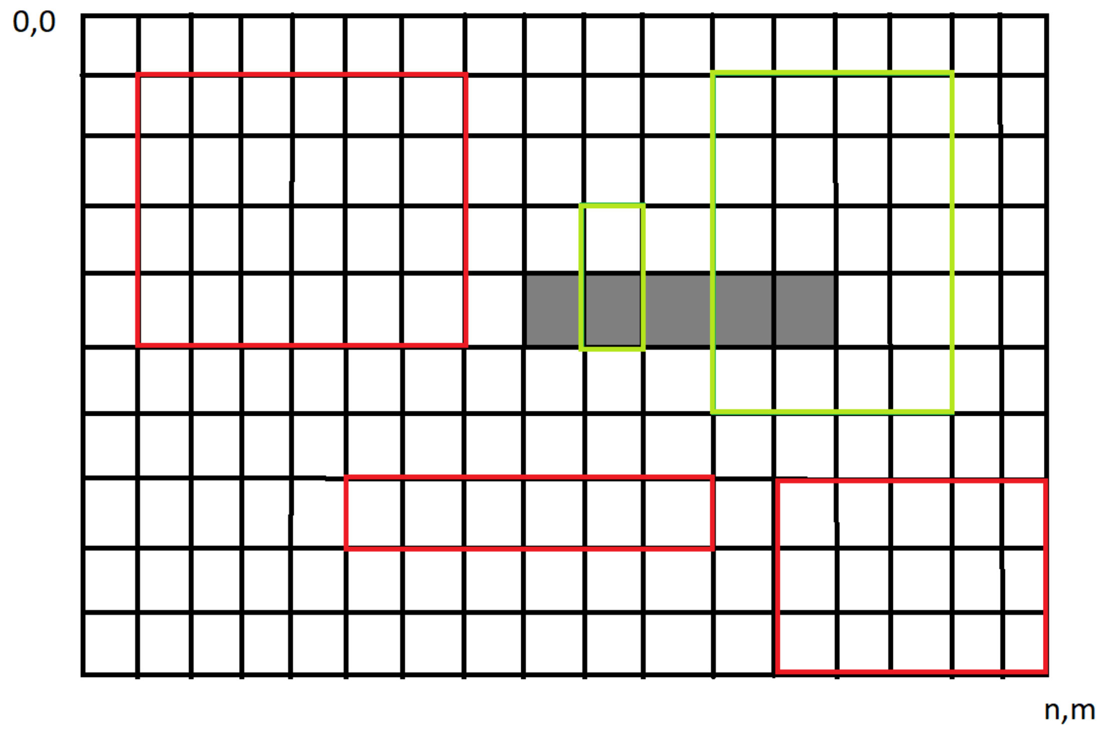
   Hence, from (a) and (b), we conclude that $M^k[i, j]$ is the number of different paths of length k from i to j by induction principle.

3. Assume that we are given a positive integer k. Design an algorithm to find the number of different paths of length k from the i-th vertex to j-th vertex for all pairs of (i, j). The time complexity of your algorithm should be $O(n^3 \log k)$. You can get partial credits if you design an algorithm of $O(n^3 k)$.

Solution: Once we have M^k for given k, we can get the number of different paths of length k from i to j for any pairs (i, j). It means that we need to design an algorithm to get M^k. As the matrix multiplication takes $O(n^3)$, we can come up with a naive algorithm of doing $k - 1$ matrix multiplications. In this case, the running time is $O(n^3 k)$. If we reduce the number of matrix multiplications, we will get a better algorithm.
Let l be the highest power of 2 less than or equal to k and p be the exponent of l. In other words, $p = \lfloor \log_2 k \rfloor$ and $l = 2^p$. To do this, let us precompute $M, M^2, M^4, M^8, ..., M^l$. Since $M^{2^i} \cdot M^{2^i} = M^{2^{i+1}}$, we can get $M^2 = M \cdot M$, $M^4 = M^2 \cdot M^2$, $M^8 = M^4 \cdot M^4$, and so on. This precomputation needs p times of matrix multiplication, so it takes $O(n^3 p)$. Next we represent k as a binary number. It has $p + 1$ digits as $2^p \leq k < 2^{p+1}$.
First, we start with the identity matrix I. If the first digit is 1, multiply it by M. Otherwise, do nothing. After that, if the second digit is 1, multiply it by M^2, otherwise, do nothing. If the third digit is 1, multiply it by M^4. Repeat this process for $p + 1$ digits. With this algorithm, we obtain M^k with at most $p + 1$ times of matrix multiplications. It also takes $O(n^3 p)$. Therefore, our algorithm takes $O(n^3 p) = O(n^3 \log k)$ to get M^k for given k.
Example: Let $k = 26$. Then, l is the highest power of 2 less than or equal to k, so it is 16, and $p = 4$. Precompute M, M^2, M^4, M^8 and M^{16}. If we represent k as a binary number, it is $11010_{(2)}$. This implies that $26 = 2^4 + 2^3 + 2^1$. Therefore, $M^{26} = M^{16} \cdot M^8 \cdot M^2$.

3. (10 pts.) You are given a grid of size $n \times m$ and within that grid there is a horizontal rod (arbitrary width but height of 1). You aim is to locate it. The only thing you can use to find the location of the rod is make queries of the form IsPresent (x_1, y_1, x_2, y_2) where $x_1 \leq x_2, y_1 \leq y_2$. This returns True if part of the rod is present in the grid from (x_1, y_1) to (x_2, y_2) and False if not. See figures below. Design an algorithm which uses queries $O(\log(n + m))$ times to find the location of the rod, i.e., the leftmost coordinates and rightmost coordinates. *Hint: consider using binary search in your algorithm.*





Solution:

1. We try to find at which row the rod is located. When making queries, we fix x_1 as 0 and x_2 as n. Then, we find the upper coordinate and the lower coordinate of the rod by varying y_1 and y_2 in the same manner as binary search. This step makes $O(\log m)$ queries.
2. We fix y_1 and y_2 in any queries later since we found the upper and lower coordinates of the rod. Then, we make queries to find the left coordinate of the rod by varying x_1 and x_2 in a binary search fashion as well. More specifically, we set x_1 as 0 and x_2 as n in the beginning, along with $prev\_x_2$ being n. Then, we start continuously making the query IsPresent (x_1, y_1, x_2, y_2). If the current query returns True, we update $prev\_x_2$ to be x_2, x_2 to be $x_1 + \lfloor \frac{x_2 - x_1}{2} \rfloor$, and make another query. On the other hand, if the current query returns False, we update x_1 to be x_2, x_2 to be $x_1 + \lfloor \frac{prev\_x_2 - x_1}{2} \rfloor$, and make another query. When the query returns True and $x_2 - x_1 \leq 1$, we find the leftmost block in the rod, and the leftmost coordinate is x_1. This step makes $O(\log n)$ queries.
3. We find the rightmost coordinate of the rod in a very similar way as step 2, though we have $prev\_x_1$ and set it to 0 in the beginning. Now, if the current query returns True, we update $prev\_x_1$ to be x_1, x_1 to be $x_1 + \lfloor \frac{x_2 - x_1}{2} \rfloor$, and make another query. If the current query returns False, we update x_2 to be x_1, x_1 to be $prev\_x_1 + \lfloor \frac{x_1 - prev\_x_1}{2} \rfloor$, and make another query. In the end where the query returns True and $x_2 - x_1 \leq 1$, we find the rightmost block instead, and the rightmost coordinate is x_2. Same as step 2, this step makes $O(\log n)$ queries.

Overall, this algorithm makes $O(\log m) + O(\log n) + O(\log n) = O(\log m + \log n) = O(\log mn)$ queries.
Lemma: $O(\log mn) = O(\log(m + n))$
Proof:

- $m + n \leq mn + mn = 2mn$ (m and n can't be positive floats that smaller than 1, based on the semantic of the problem). So, $\log(m + n) = O(\log mn)$.

- $mn \leq (m+n)^2 = m^2 + 2mn + n^2$. So, $\log mn = O(\log(m+n))$.
- Consequently, $\log mn = \Theta(\log(m+n))$ and $O(\log mn) = O(\log(m+n))$.

From the lemma, our algorithm indeed makes $O(\log mn) = O(\log(m+n))$ queries.

Rubrics:

Problem 1, 15 pts

1. (a) 3 points : proved/explained 1.a correctly.
   (b) 3 points : proved/explained 1.b correctly.
2. 5 points : proved/ explained region correctly.
3. 3 points : partially correct explanation of region.
4. 4 points : proved that l^* is in the region s^*
5. 2 points : partially correct explanation of l^* is in the region s^*.

Problem 2, 15 pts

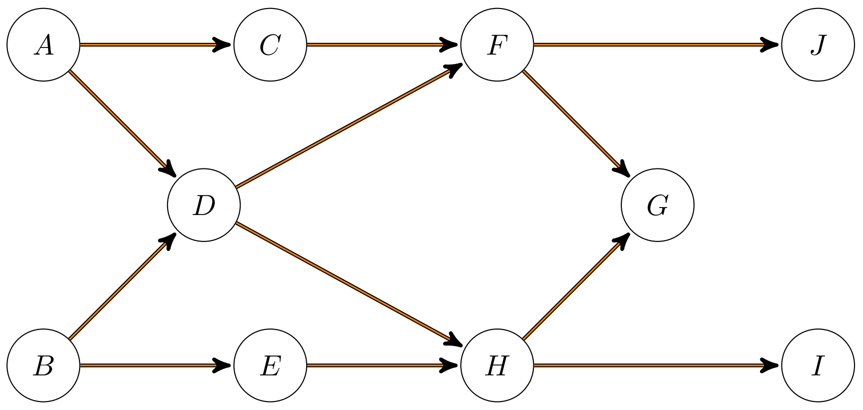
1. (a) 3 points : Provided a correct explanation or proof.
2. (a) 6 points : Provided a correct proof.
   (b) 3 points : Provided an incomplete proof or explanation
3. (a) 6 points : Provided an algorithm in $O(n^3 \log k)$
   (b) 4 points : Provided an algorithm in $O(n^{\log_2 7} k)$ ($n^{\log_2 7}$ may come from the optimized matrix multiplication in the textbook)
   (c) 3 points : Provided an algorithm in $O(n^3 k)$
4. 1.5 point : I don't know how to answer this question.

Problem 3, 10pts

- 3 points: Described a correct way to find at which row the rod is i.e. the uppermost and lowermost coordinates.
- 4 points: Described a correct way to find the leftmost and rightmost coordinates.
- 3 points: Provided a reasonable running time analysis for the algorithm, and it runs in $O(\log(m+n))$ or $O(\log(mn))$.
- 1 point: I don't know how to answer this question.

Solution 05

1. (10 pts.) Consider the following directed graph.



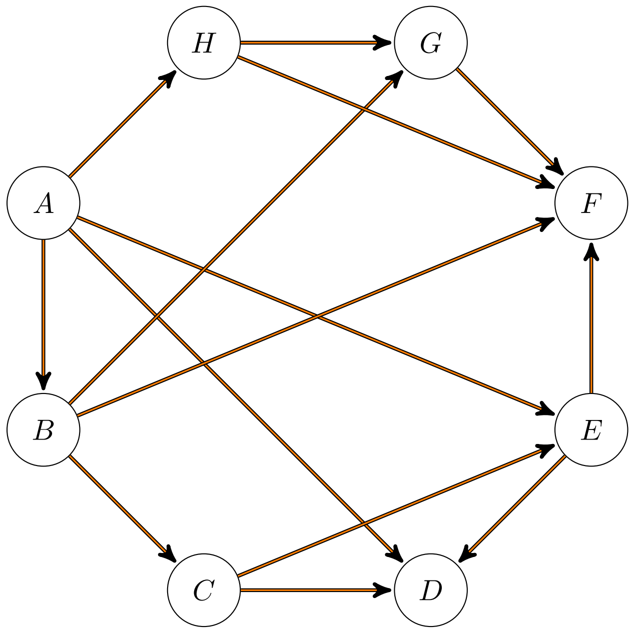
1. What are the sources and sinks of the graph?
2. Give one linearization of this graph.
3. How many linearization does this graph have?

Solution:

1. Sources of a graph do not have any in-edges. So, A and B are the sources. Sinks do not have any out-edges. So J, G and I are the sinks.
2. $(A, B, C, D, E, F, H, G, I, J)$ is one of linearizations.
3. It has 356 linearizations. Consider F and H; first assume that F is before H in the linearization. What are the possible situations between F and H in the linearization? There are 5 cases:
   (a) FH. Then A, B, C, D, E are before it and G, J, I are after it. It is not hard to calculate that the number of linearizations for the former is 16 and the number of linearization for the latter is 6.
   (b) FEH. Then A, B, C, D are before it and G, J, I are after it. The number of linearizations for the former is 5 and the number of linearization for the latter is 6.
   (c) FJH. Then A, B, C, D, E are before it and G, I are after it. The number of linearizations for the former is 16 and the number of linearization for the latter is 2.
   (d) $FEJH$. Then A, B, C, D are before it and G, I are after it. The number of linearizations for the former is 5 and the number of linearization for the latter is 2.
   (e) $FJEH$. Then A, B, C, D are before it and G, I are after it. The number of linearizations for the former is 5 and the number of linearization for the latter is 2.

   Summing them up gives $16 \cdot 6 + 5 \cdot 6 + 16 \cdot 2 + 5 \cdot 2 + 5 \cdot 2 = 178$. Don't forget this is under the assumption that F is before H, the other way around is symmetric. So the total number of linearization is $178 \cdot 2 = 356$.

2. (10 pts.) Consider the following directed graph.



1. Perform depth-first search with timing (DFS-with-timing) on the above graph; whenever there's a choice of vertices, pick the one that is alphabetically first: give the pre and post number of each vertex.
2. Draw the meta-graph of this graph, and give the vertices in each connected component.

Solution:

1. A: 1,16
   B: 2,13
   C: 3,10
   D: 4,5
   E: 6,9
   F: 7,8
   G: 11,12
   H: 14,15
2. As there are no cycles in the graph, meta graph of the given graph is itself.

3. (10 pts.) You are in charge of the United States Mint. The money-printing machine has developed a strange bug: it will only print a bill if you give it one first. If you give it a d-dollar bill, it is only willing to print bills of value $d^2 \mod 400$ and $d^2+1 \mod 400$. For example, if you give it a \$6 bill, it is willing to print \$36 and \$37 bills, and if you then give it a \$36-dollar bill, it is willing to print \$96 and \$97.

You start out with only a \$1 bill to give the machine. Every time the machine prints a bill, you are allowed to give that bill back to the machine, and it will print new bills according to the rule described above. You want to know if there is a sequence of actions that will allow you to print a \$20 bill, starting from your \$1 bill. Model this task as a graph problem: give a precise definition of the graph (what are the vertices and edges) involved and state the specific question about this graph that needs to be answered. Give an algorithm to solve the stated problem and give the running time of your algorithm.

Solution: $G = (V, E)$.

V: we have at most 400 possible bills (includes \$0 bill). We add 400 vertices $V = \{v_1, v_2, ..., v_{400}\}$ to represent these 400 possible bills (e.g. v_1 represents \$1 bill, v_5 represents \$5 bill, and v_{400} represent \$0 bill which can be generated by giving \$20).

E: we enumerate all vertices to add edges. For each vertices v_k, we have $i = k^2 \mod 400$ and $j = k^2 + 1 \mod 400$, so we add a directed edge from v_k to v_i and a directed edge from v_k to v_j.

Then the problem becomes if there is a path to reach v_{20} from a given vertex v_1 in the built graph G.

We can apply the *explore* algorithm start from v_1 to solve this problem. The running time for applying *explore* algorithm is $O(|V| + |E|)$. The running time for creating vertices and edges is $O(|V| + |E|)$. So the total running time is $O(|V| + |E|)$.

Rubrics:

Problem 1, 10pts

- 2 points : Provided the correct answer.
- 3 points : Provided a correct linearization.
- 5 points(Bonus) : Provided the correct answer.
- 1 point: I don't know how to answer this question

Problem 2, 10pts

- 6 points : Provided the correct pre an post numbers for all nodes.
- 3 points : Provided a correct pre and post number but didn't follow alphabetical order when there is a choice of vertices.
- 4 points : identified that metagraph is the given graph itself.(no need to draw the graph).
- 1 point: I don't know how to answer this question

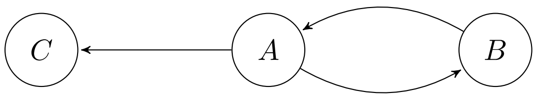
Problem 3, 10pts

- 5 points: construct the graph correctly
  - 2 points : add vertices correctly
  - 3 points : add edges correctly
- 5 points: correct algorithm and running time analysis
  - 1 points : transfer the problem correctly (e.g. check if there is a path from v_1 to v_{20})
  - 3 points : apply explore algorithm
  - 1 points : appropriate running time analysis
- 1 point: I don't know how to answer this question

Solution 06

1. **(10 pts.)** Consider this algorithm to find all connected components of a directed graph G: run DFS-with-timing on G to get the postlist (i.e., the list of vertices in decreasing post value), and then use the reverse of the postlist as a "magic ordering" to run the DFS algorithm on graph G. Design an instance to demonstrate that this algorithm is incorrect. Specifically, you will need to give a directed graph, then run above algorithm and show that the resulting $visited$ array does not give the correct connected components of G.

   Solution: Consider this graph G.

   DFS-with-timing on G with the alphabetical order gives us the postlist (A, C, B). If we redo DFS with the reverse of the postlist (B, C, A) as a "magic ordering", it will start DFS with B and the resulting $visited$ array is an array of all 1's. It means that $\{A, B, C\}$ is a connected component, which is not a true.

2. **(10 pts.)** Given a directed graph $G = (V, E)$, a vertex $t \in V$, and an edge $e = (u, v) \in E$, design an $O(|V| + |E|)$ time algorithm to determine whether there exists a cycle in G that contains both t and e.

   Solution: We have that there exists a cycle in G that contains both t and e if and only if there exists a path from v to t and then from t to u in G. Therefore, we can design the following algorithm.

   Algorithm. We run explore twice. The first run decides whether there exists a path from v to t in G. The second run decides whether there exists a path from t to u in G. For the first run, we run explore (G, v). If in the resulting visited array we have visited[t] = 1, (i.e., t can be reached from v in G), then we continue the second run; otherwise the algorithm terminates and return that such cycle does not exist. For the second run, we run explore (G, t). If in the resulting visited array we have visited[u] = 1, (i.e., u can be reached from t in G), then the algorithm returns that such cycle exists; otherwise there does not exist such a cycle.

   Running Time. Initiation of visited array runs $O(|V|)$ time. Run explore (G, v) and explore (G, t) both take $O(|E|)$ time. Therefore, the entire algorithm runs in $O(|V| + |E|)$ time.

3. **(10 pts.)** Design an algorithm runs in $O(|V| + |E|)$ time which takes a directed graph as the input and determines if there is a vertex such that all other vertices are reachable from it.

   Solution:
   We can make the following observations before designing an algorithm:

   - **If such vertex exists, it must be in the connected component that is a source in the meta-graph** (which we'll call source connected component from now on). If it's in any non-source connected component, it wouldn't be able to reach to any vertex in a source connected component which contradicts with the requirement that all other vertices are reachable from that particular vertex.
   - **There can only be one source connected component.** If there are multiple source connected components, a vertex in one source connected component wouldn't be able to reach a vertex in another source connected component which contradicts with the requirement as well.

   With these observations, the algorithm simply needs to check if there is only one source connected component. There are multiple ways to do that:

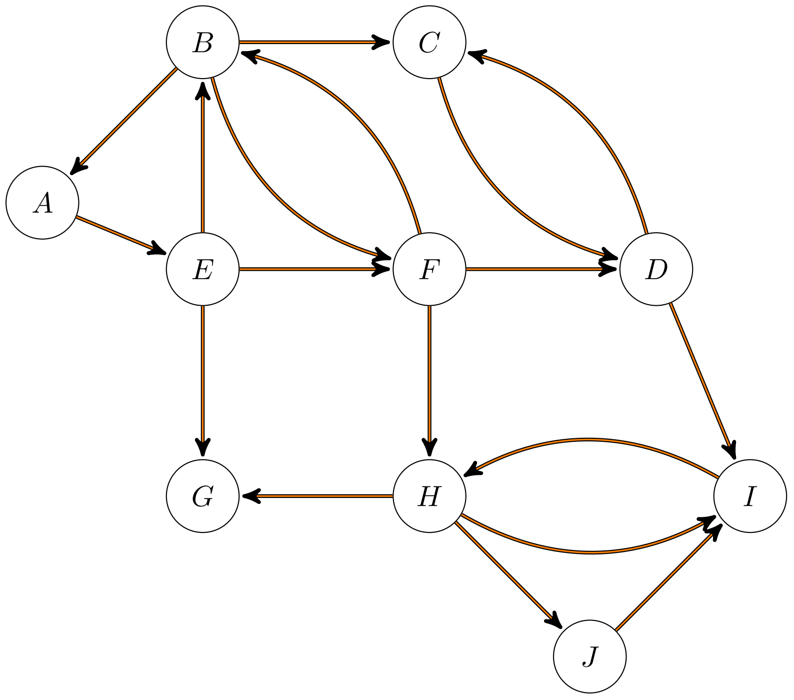
- Possible algorithm 1
  1. Based on Claim 1 in lecture A11 typed note and the fact that any meta-graph is a DAG, we know that the vertex with the largest post number is in a source connected component after running DFS-with-timing. So, we do this and get that vertex i.e. the first vertex in the postlist. Time complexity: $\Theta(|V| + |E|)$. DFS-with-timing simply records pre, post numbers for each vertex and creates a postlist as it does DFS, so it still has the same running time as the regular DFS.
  2. Run *explore* on that vertex. If all other vertices are visited in the procedure, the algorithm returns True. Otherwise, we know there is certain connected component that the source connected component containing that vertex can't reach. Since any meta-graph is a DAG, there must be at least one source connected component. Thus, it must be the case where there are multiple source components, and the algorithm can safely return False. Time complexity: $O(|V| + |E|)$.

  Overall, the running time is $O(|V| + |E|)$.
- Possible algorithm 2
  1. Run the special version of DFS used to obtain all conncected components i.e. the algorithm mentioned in page 4 of lecture A11 typed note. Time complexity: $\Theta(|V| + |E|)$.
  2. Go through array *visited* to find the vertices assigned with the largest $num\text{-}cc$. They consist of a source connected component. Time complexity: $\Theta(|V|)$.
     Correctness: The connected component containing these vertices is the last connected component we found. This implies that if we find all connected components by iteratively removing a sink connected component in the graph (as mentioned in page 3 of lecture A10 typed note), this particular connected component would be the last connected component remaining. Essentially, it is a source connected component since all non-source connected component would be removed before a source connected component.
  3. Same as step 2 in possible algorithm 1. Pick any vertex we found in the previous step and run *explore* on that vertex.

  Overall, the running time is $O(|V| + |E|)$.
- Possible algorithm 3
  1. Same as step 1 in possible algorithm 2.
  2. Construct the meta-graph for the given graph. More specifically, $V_m = \{C_1, \ldots, C_n\}$, assuming $num\text{-}cc = n$ in the end. For all $(u, v) \in E$, add (C_u, C_v) to E_m, assuming C_u is the connected component containing u and C_v is the connected component containing v. Time complexity: $\Theta(|E|)$. Note that we don't need to go through array *visited* or V.
  3. Go through E_m to find the in-degree of all vertices in the meta-graph. The vertices with in-degree 0 represent source connected components. If there is only one vertex with in-degree 0, the algorithm returns True. Otherwise, it returns False. Time complexity: $O(|E|)$ since $|E_m| \leq |E|$.

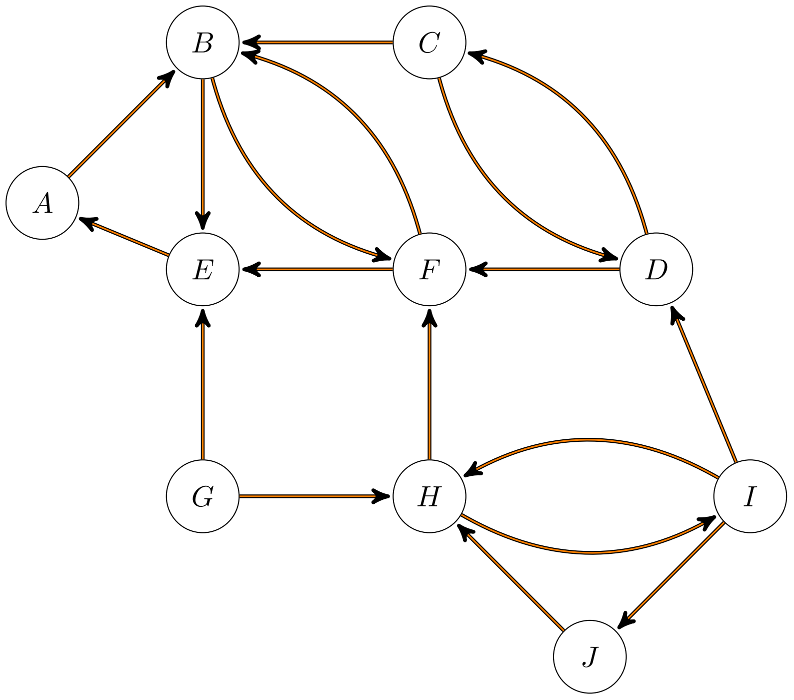
  Overall, the running time is $O(|V| + |E|)$.

4. (12 pts.) Run the strongly connected components algorithm on the following directed graphs G. When doing DFS on G^R: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.

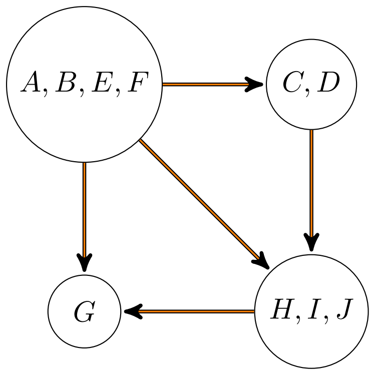


1. Give the pre and post number of each vertex in the reverse graph G^R.
2. In what order are the connected components found?
3. Which are source connected components and which are sink connected components?
4. Draw the "metagraph" (each meta-node is a connected component of G).
5. What is the minimum number of edges you must add to G to make it strongly connected?

Solution: Reverse graph G^R is as follows,



1. pre and post numbers in G^R
   A : 1, 8
   B : 2, 7
   C : 9, 12
   D : 10, 11
   E : 3, 4
   F : 5, 6
   G : 13, 20
   H : 14, 19
   I : 15, 18
   J : 16, 17
2. arranging the vertices in the decreasing order of post numbers,
   G, H, I, J, C, D, A, B, F, E
   Performing DFS on the given graph with above order will give us the connected components as follows,
   $\{$G$\}$, $\{$H, I, J$\}$, $\{$C, D$\}$, $\{$A, B, E, F$\}$
3. Source connected components: $\{$A, B, E, F$\}$
   Sink connected components: $\{$G$\}$
4. Meta graph of the given graph is as follows,



5. A graph is (strongly) connected means the meta-graph consists of a single vertex. Therefore, we should add edges from all sink components to all source components. Specifically, we can pick one vertex u in a sink component and pick one vertex v in a source component and add an edge (u, v), and we should do this for every pair of sink component and source component. In this scenario there is only one sink component and one source component, So making an edge from G to one of the four vertices in $\{A, B, C, D\}$ will make the resulting graph connected.

Rubrics:

Problem 1, 10pts

- 5 points : Provided an appropriate counterexample, especially, the graph should be directed.
- 5 points : Provided an explanation with the postlist after the first DFS and the resulting $visited$ array.
- 1 point : I don't know how to answer this question.

Problem 2, 10pts

- 10 points : correct algorithm
  - 3 points : appropriate explanation about twice explore
  - 5 points : apply explore algorithm/DFS
  - 2 points : correct return statements (e.g. return true if both b1 and b2 are true; otherwise return false) and running time analysis.
- 1 point : I don't know how to answer this question

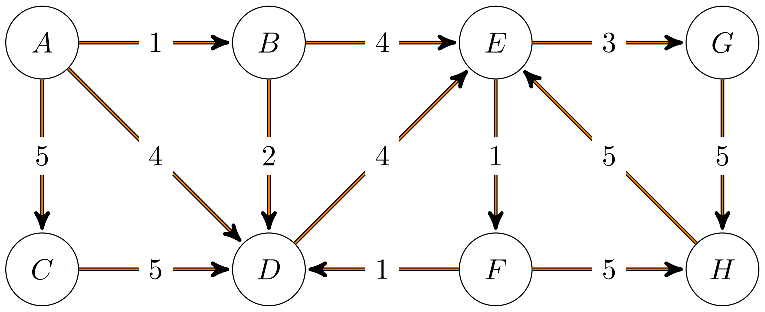
Problem 3, 10pts

- 2 points: Found out such vertex must be in a source connected component if it exists.
- 2 points: Found out the question is asking to find whether or not there is only one source connected component.
- 3 points: Provided a correct algorithm.
- 1 point: The provided algorithm has running time $O(|V| + |E|)$.
- 2 points: Provided a correct running time analysis.
- 1 point : I don't know how to answer this question.

Problem 4, 12pts

- 3 points : correct pre and post numbers of the reverse graph
- 3 points : correct connected components and their order.
- 1 point : correct source connected component ($\{$A, B, E, F$\}$).
- 1 point : correct sink connected component ($\{$G$\}$).
- 2 points : correct metagraph with 4 nodes and 5 edges.
- 1 point : one or more edges missing in the metagraph
- 2 point : identified that one edge need to be added (from sink to source).
- 1.2 point : I don't know how to answer this question.

1. **(8 pts.)** Run Dijkstra's algorithm on the following graph, starting at node A. Whenever there is a choice of vertices with same $dist$ value, always pick the one that is alphabetically first. Specifically, you are asked to draw a table in which each row shows the $dist$ array at each iteration of the algorithm.



Solution: Lets run the Dijkstra's algorithm on the given graph G(V,E)

Itr	A	B	C	D	E	F	G	H
0	0	∞	∞	∞	∞	∞	∞	∞
1	**0**	1	5	4	∞	∞	∞	∞
2	**0**	**1**	5	3	5	∞	∞	∞
3	**0**	**1**	5	**3**	5	∞	∞	∞
4	**0**	**1**	**5**	**3**	5	∞	∞	∞
5	**0**	**1**	**5**	**3**	**5**	6	8	∞
6	**0**	**1**	**5**	**3**	**5**	**6**	8	11
7	**0**	**1**	**5**	**3**	**5**	**6**	**8**	11
8	**0**	**1**	**5**	**3**	**5**	**6**	**8**	**11**

2. **(10 pts.)** There is a country of n islands. m bridges are installed between some of these islands allowing us to travel in both directions. We have two factories on distinct islands and need to transport goods between theses two factories. However, since each bridge has a weight limit, if an amount of goods exceeding the weight limit passes though the bridge, the bridge will collapse. We know that the weight limit of each bridge is at most w; you may assume that w is an integer. Design an algorithm to find the maximum weight of the goods that can be moved in one transportation. The time complexity of your algorithm should be $O((n+m)\log w)$. You can get partial credits if you design an algorithm of $O((n+m)w)$.

Solution: We can build a graph using islands as vertices and bridges as edges. Suppose two factories are in islands A and B, we will load goods in the factory on island A and then move them to the one on island B. Let M be the maximum weight of the goods can be moved from A to B in one transportation. We know there are $w+1$ possible number of maximum weight $\{0,1,2,...w\}$. We can use binary search to find M. For a particular maximum-weight mid, we can test whether there exists a path from A to B on which all edges have limit at least mid (i.e., goods of weight mid can be transported from A to B) by first removing all edges whose weight limit is below mid and then testing if A can reach B in the resulting graph using BFS or DFS.

Step 1: construct the graph G using islands as vertices and bridges as edges.
Step 2: initialize four integers, $low = 0, high = w, mid = 0, max\_w = 0$.

Step 3: update mid, $mid = (low + high)/2$; make a copy of G and name it as G'; then update the adjacency list of G' to remove any edge whose weight is less than mid.
Step 4: run BFS/DFS to find if there is a path from A to B in G'. If yes, it means M is at least mid, so we have $max\_w = mid$ and $low = mid + 1$; otherwise, it means M is less than mid, so we have $high = mid - 1$.
step 5: while $low \leq high$, repeat steps 3–4. Return max_w after finishing the while loop.

Running time: time complexity of step 1 is $O(n + m)$, time complexity of step 2 is $O(1)$, time complexity of step 3 is $O(m)$, time complexity of step 4 is $O(n + m)$. We may need to run multiple times of step 3 and step 4, and the number of times is $O(\log w)$ since we are doing binary search and in each iteration $(high - low)$ will be halved. So the total running time is $O((n + m) \log w)$.

3. (10 pts.) For a given directed graph $G = (V, E)$, let us denote $V = \{1, 2, ..., n\}$. Define a function $D(G, s, t)$ that gives the distance from s to t.

1. We want to get the length of the shortest path from 1 to n that must pass through a vertex v. Prove that $D(G, 1, v) + D(G, v, n)$ gives the desired quantity. You may use proof by contradiction.
2. We want to get the length of the shortest path from 1 to n that must pass through two vertices v and w. Using the above, find a formula to represent the quantity in terms of the function D and vertices $1, n, v, w$.

Solution:

1. Suppose that there exists a path p from 1 to n passing through v with distance $d < D(G, 1, v) + D(G, v, n)$. We can divide p into two parts p_1 and p_2 so that p_1 is a path from 1 to v and p_2 is a path from v to n. Let d_1 and d_2 be the distance from 1 to v via p_1 and from v to n via p_2, respectively. Then, $d = d_1 + d_2 < D(G, 1, v) + D(G, v, n)$. If $d_1 < D(G, 1, v)$, it is a contradiction because $D(G, 1, v)$ returns the shortest path from 1 to v so it must count p_1 during the function as well. Therefore, $d_1 \geq D(G, 1, v)$. Similar to this, we also obtain that $d_2 \geq D(G, v, n)$. Combining these two inequalities gives us $d_1 + d_2 \geq D(G, 1, v) + D(G, v, n)$, which is a contradiction. Therefore, there is no path from 1 to n passing through v with a shorter distance than $D(G, 1, v) + D(G, v, n)$
   We also need to make sure that $D(G, 1, v) + D(G, v, n)$ is indeed the shortest path from 1 to n passing through a vertex v. $D(G, 1, v)$ returns the shortest path from 1 to v, so we can make sure that there exists a path p_1 from 1 to v with the distance $D(G, 1, v)$. The same goes for the case of v and n, denote it as p_2. Then, by concatenating p_1 and p_2, we obtain the desired path with distance $D(G, 1, v) + D(G, v, n)$.
2. If we must pass two vertices v and w, there are two possible paths of $1 \to v \to w \to n$ and $1 \to w \to v \to n$. Using the above, we figure out that the corresponding shortest paths are $D(G, 1, v) + D(G, v, w) + D(G, w, n)$ and $D(G, 1, w) + D(G, w, v) + D(G, v, n)$, respectively. Therefore, we can obtain the desired shortest path by $\min\{D(G, 1, v) + D(G, v, w) + D(G, w, n), D(G, 1, w) + D(G, w, v) + D(G, v, n)\}$

4. (8 pts.) You are given a directed graph $G = (V, E)$ and a vertex $s \in V$. Each edge e is assigned with a length $l(e)$, possibly with negative value. We know that there is no negative cycle in this graph, and that the only negative edges are the ones that leave the vertex s. That is, $l(s, v) < 0$ for all $(s, v) \in E$, and $l(u, v) > 0$ for all $u \neq s$ and $(u, v) \in E$. If we run Dijkstra's algorithm starting at s, will it fail on this graph? Prove your conclusion.
Solution: No, it won't fail on this graph.
Proof: The correctness of Dijkstra's algorithm depends on the claim that the next closes vertex, i.e., v^*_{k+1},

must be within one-edge extension of R_k i.e. $v^*_{k+1} = \arg\min_{v \notin R_k, u \in R_k, (u,v) \in E}(distance(s,u) + l(u,v))$. The proof of this statement *only* uses that, the edges *leaving* any $v \notin R_k$ have positive edge length. In other words, the proof *only* requires that the one-edge extension is always preferred than the two-edge extension (first edge being (u,v), second edge being (v,w) for some w) will always be longer since the second-edge has positive edge length. In our case, the second-edge always has positive length. This is because, although edges leaving s have negative edge length, s will always stay in R_k and consequently edges leaving s will always be part of one-edge extension.

Rubrics:

Problem 1, 8pts

1. 8 pts : All the iterations and final answer are correct
2. 6 pts : Didn't follow alphabetical order(i.e., the final answer is correct the choice of vertices in the iterations are different).
3. 3 pts : more than half of the final shortest paths(¿4) are correct.
4. 0.8 pt : I don't know how to answer this question.

Problem 2, 10pts

1. - 10pts : correct algorithm with $O((n+m)\log w)$ running time.
   - 7pts : correct algorithm with $O((n+m)w)$ running time.
   - 5pts : incorrect algorithm but apply path finding algorithms (BFS/DFS).

   1pt : I don't know how to answer this question.

Problem 3, 10pts

1. - 3pts : Proved there is no desired path with a shorter distance than $D(G,1,v)+D(G,v,n)$.
   - 3pts : Proved $D(G,1,v)+D(G,v,n)$ is indeed the desired shortest path.
2. - 4pts : Provided a correct formula.
   - 2pts : Provided one of them.

   1pt : I don't know how to answer this question.

Problem 4, 8pts

- 3pts: Correct conclusion.
- 5pts: Provided a proof that makes sense.
- 0.8pts: I don't know how to answer this question.