Advanced Divide & Conquer

Fast Algorithm for Power of Matrix

Divide and Conquer on Trees

Section 1

Fast Algorithm for Power of Matrix

Calculating a^n: Fast Algorithm

Consider the simple question: calculate a^n, where a and n are positive intergers. Assume any single multiplication takes constant time[1].
Naive approach requires $O(n)$ multiplication. But what if n is super large, say, 10^{18}?
We introduce a simple $O(\log n)$ algorithm.

[1]In reality, only the multiplication of small integers take constant time. However, for example, if we are instead interested in a^n mod P, where P is less than 2^{31}, the assumption is actually valid.

The Algoritm

```
Data: Integer $a$ and $n$.
Result: An integer, which is $a^n$.
begin
    if $n = 0$ then
        return 1
    else if $n = 1$ then
        return $a$
    else if $n$ is even then
        return $(\texttt{Power}(a, n/2))^2$
    else
        return $\texttt{Power}(a, n-1) \times a$
    end
end
```

Algorithm 1: Power

Time Complexity

We claim that the complexity of the algorithm is $O(\log n)$.

Let $f(n)$ denote the running time of Power(a, n), for the fixed a. If $n \leq 1$, $f(n) = O(1)$. Otherwise, if n is even, then $f(n) = f(\frac{n}{2}) + O(1)$, and if n is odd, then $f(n) = f(n-1) + O(1) = f(\frac{n-1}{2}) + O(1)$.

Therefore, for $n \geq 2$, $f(n) \leq f(\lfloor \frac{n}{2} \rfloor) + O(1)$. It is immediate that $f(n) = O(\log n)$.

Generalization: Sum of Geometric Series of Matrix

The algorithm is easy to be applied on the matrices, that is, we can calulate M^n in $O(m^\omega \log n)$, where M is an $m \times m$ matrix, and m^ω is the complexity of a single matrix multiplication.

Next question: How to compute $M + M^2 + M^3 + \ldots + M^n$ efficintly? Suppose now we have $g(M, n)$ that can compute M^n in time $m^\omega \log n$.

Generalization: Sum of Power Series of Matrix

Consider the procedure below.

Define $f(M, n)$ to be $M + M^2 + M^3 + \ldots + M^n$. If $n = 1$, $f(M, n) = M$. If $n = 2$, $f(M, n) = M + M^2$.

For general n, if n is even, then $f(M, n) = f(M, \frac{n}{2}) \times (I + g(M, \frac{n}{2}))$, and if n is odd, then $f(M, n) = f(M, n-1) + g(M, n)$.

It is easy to see that the time compexity is $O(\log^2 n)$
You can practice this problem online at `http://poj.org/problem?id=3233`.

Improved Recursion for Sum of Power Series of Matrix

We actually have an algorithm running in $O(\log n)$ time. The observation is that we can calculate the sum of power series and M^n simultaneously.

Define $f(M, n)$ to be the pair $(M + M^2 + M^3 + \ldots + M^n, M^n)$. If $n = 1$, $f(M, n) = (M, M)$. If $n = 2$, $f(M, n) = (M + M^2, M^2)$. For general n, if n is even, let $(A, B) = f(M, n/2)$ and $f(M, n) = (A(A + I), B^2)$; otherwise n is odd, let $(A, B) = f(M, n - 1)$ and $f(M, n) = (A + BM, BM)$.

Application: Linear Recursion

Suppose we have $a_{n+2} = A \cdot a_{n+1} + B \cdot a_n$, $a_0 = x$, $a_1 = y$. How to calculate a_n, in $O(\log n)$ time?

Observation: $\begin{pmatrix} a_{n+1} \\ a_{n+2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ B & A \end{pmatrix} \begin{pmatrix} a_n \\ a_{n+1} \end{pmatrix}$

Then, $\begin{pmatrix} a_n \\ a_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ B & A \end{pmatrix}^n \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$

We convert the linear recursion into a matrix multiplication problem. Similarly, we can find the sum $\sum_{i=1}^{n} a_i$ in $O(\log^2 n)$ matrix multiplications.

Also note that the dimension of the matrix is the order of the linear recursion, and we can actually solve linear recursion of any order. You can solve this problem online at `http://acm.hdu.edu.cn/showproblem.php?pid=1005`.

Application: Vertices Visited in K Hops

Given a directed graph $G = (V, E)$, calculate the number of different ways to visit each vertex by exactly K hops. A way to visit a vertex is a sequence of vertices consisting of exactly K elements, where each two consecutive vertices denote an edge in G.

Suppose M is the adjacent matrix of G. Then, consider $S = M^K$. We can show that S_{ij} is the number of ways going from i to j.

You can practice a slightly more difficult problem at `http://poj.org/problem?id=3613`.

Section 2

Divide and Conquer on Trees

Median Decomposition

Given an edge weighted tree T, calculate the number of vertex pairs (u, v) such that $d(u, v) \leq k$, where $d(u, v)$ is the distance on the tree in terms of the weights, and k is some given constant.

Naive approach gives $O(|T|^2)$ running time.

We introduce a divide and conquer approach in which the running time is $O(|T| log^2 |T|)$.
After you learn the technique, you can test your understanding at `http://poj.org/problem?id=1741`.

Median Decomposition: Divide

We describe the divide and conquer framework in this problem. For some fixed vertex r, suppose we pick r as root and the subtrees rooted at the children of r are $T_1, T_2, \ldots, T_m$. The procedure for picking r is described later, and we assume that r can be picked in $O(|T|)$ time.

For each T_i, we solve the problem recursively. Observe that now the pairs (u, v) such that u, v are in the same T_i are solved. It remains to deal with the pairs that are in the different T_i's.

Median Decomposition: Conquer

Suppose u is in T_i and v is in T_j. Obviously, the path from u to v visits r. Since it's a tree, $d(u, v) = d(u, r) + d(v, r)$.

To avoid the n^2 calculation, we first use DFS to calculate the distance from any vertex to the root r. Then, the problem reduces to find (u, v), such that $u \in T_i$, $v \in T_j$, for $i \neq j$, $d(u, r) + d(v, r) \leq k$.

Observe that the number of pairs (u, v) in *different* subtrees such that $d(u, v) \leq k$, equals the *total* number of pairs (u', v') in T such that $d(u, v) \leq k$, minus the number of pairs (u'', v'') that are in the *same* subtree T_i for all i, such that $d(u'', v'') \leq k$.

Median Decomposition: Counting

To calculate the number of pairs (u, v) in T such that $d(u, r) + d(v, r) \leq k$, we sort all the vertices u in T with respect to $d(u, r)$. Then in $O(|T| \log |T|)$ time, we can calculate the number of pairs (u, v) such that $d(u, r) + d(v, r) \leq k$.

We can then sort each T_i and count the pairs of vertices that are both in T_i, using the similar approach as in the above, in $O(|T| \log |T|)$ time.

Then we have the number of pairs that are in the different T_i's, such that the distance between them is at most k.

Median Decomposition: Analysis of Divide and Conquer

Let $f(T)$ denote the running time required for instance T. Then the divide part takes $\sum_{i=1}^{m} f(T_i) + O(|T|)$. In the conquer part, we sort and count in time $O(|T| \log |T|)$ (assuming merge sort is used).

Therefore, $f(T) \leq \sum_{i=1}^{m} f(T_i) + C \cdot |T| \log |T|$, where C is some universal constant.

Median Decomposition: Analysis of Divide and Conquer

Observe that the final complexity depends on how we pick r.

Define the subtrees obtained by removing r and its adjacent edges to be $T_1, T_2, \dots T_c$. We claim that there exists r such that for all $1 \le i \le c$, $|T_i| \le \lfloor \frac{|T|}{2} \rfloor$. We prove the claim later. Here, we first show if we pick such r, the complexity is $O(|T| \log^2 |T|)$.

Recall that $f(T) \le \sum_{i=1}^{m} f(T_i) + C \cdot |T| \log |T|$. We prove by induction on $|T|$, that $f(T) \le C \cdot |T| \log^2 |T|$. In the base case when $|T| = 1$, the claim holds trivially.

For general T,

$$\begin{aligned}
f(T) &\leq \sum_{i=1}^{m} f(T_i) + C \cdot |T| \log |T| \\
&\leq \sum_{i=1}^{m} C|T_i| \log^2 |T_i| + C \cdot |T| \log |T| \\
&\leq \sum_{i=1}^{m} C|T_i| \log |T| \log \frac{|T|}{2} + C \cdot |T| \log |T| \\
&< C|T| \log |T|(\log |T| - \log 2) + C|T| \log |T| = C|T| \log^2 |T|
\end{aligned}$$

Median Decomposition: Picking Right r

Definition

Suppose we have a tree T. For vertex u, define $T_1^{(u)}, T_2^{(u)}, \ldots, T_{c_u}^{(u)}$ to be the subtrees obtained by removing u and its adjacent edges. Define $f(u) = \max_{1 \le i \le c_u} |T_i^{(u)}|$. The median of the tree is defined as $\arg\min_{u \in T} f(u)$.

Theorem

Suppose r is (any of) the median of tree T, and the subtrees obtained by removing r and its adjacent edges are $T_1, T_2, \ldots, T_c$. Then for any $1 \le i \le c$, $|T_i| \le 1 + \sum_{j : j \ne i, 1 \le j \le c} |T_j|$. Note that this implies that $|T_i| \le \lfloor \frac{|T|}{2} \rfloor$.

Proof.

We suppose for contradiction that there exists i, such that $|T_i| > 1 + \sum_{j:j\neq i, 1\leq j\leq c} |T_j|$. Define $r' \in T_i$ to be the vertex adjacent to r (Observing that r' is uniquely defined). We show that $f(r') < f(r)$, which leads to a contradiction.

First, we know that $f(r) = |T_i|$, since otherwise if there is a j such that $|T_j| > |T_i|$,
$2|T_i| < |T_i| + |T_j| \leq |T| = 1 + \sum_{j=1}^{c} |T_j| < 2|T_i|$, contradiction.

Then, the subtrees obtained by r' contains

- Subtrees with vertex set that is a subset of T_i. These vertex sets are of cardinality strictly less than $|T_i|$.
- Subtree with vertex set $S = \{r\} \cup (\cup_{j:j\neq i, 1\leq j\leq c} T_j)$. We note that $|S| = 1 + \sum_{j:j\neq i, 1\leq j\leq c} |T_j| < |T_i|$.

Therefore, $f(r') < f(r)$, contradiction. □

Median Decomposition: Finding The Median

We show that we can find the median in $O(|T|)$ time, using the following procedure.

1. Pick any vertex to be root. Define the children set of u for all $u \in T$ to be $C(u)$.
2. Define $s(u)$ to be the number of vertices in the subtree rooted at u, for $u \in T$. We calculate $s(u)$ by DFS, using the recursion that $s(u) = 1 + \sum_{v \in C(u)} s(v)$.
3. Find $r \in T$ such that $\max\{\max_{v \in C(r)} s(v), |T| - s(r)\}$ is minimized. Return r as the median.

It is immediate that the procedure finds the median, and the first two steps take $O(|T|)$ time. In the third step, for each vertex u in T, it takes $O(|C(u)|)$ time, so the total time is $\sum_{u \in T} O(|C(u)|)$, which is at most constant times the number of edges of T. Therefore, the running time is linear in $|T|$.

Median Decomposition

Combining the above, we recall the whole algorithm ALG(T):

1. Find r as the median of T.
2. Suppose the subtrees obtained by removing r and its adjacent edges to be $T_1, T_2, \dots, T_m$. Solve the sub-problems by ALG(T_i).
3. Calculate the number of pairs (u, v) such that u and v are in different subtrees that satisfy $d(u, v) \leq k$.
4. Combine the results in step 2 and 3.

The whole algorithm, as has been analyzed, runs in $|T| \log^2 |T|$ time.

Heavy-light Decomposition

We consider the problem that doing statistics on the path of trees.

Given an edge weighted tree T with n vertices, we are to maintain a structure that supports:

- Given vertex u and v, set the weights of each edges on the path from u to v to be c.
- Given vertex u and v, query the sum of weights of edges on the path from u to v.

We note that if T is a line, then the problem can be solved using Segment Trees.

We introduce a way to partition the edges of the tree into paths and other edges, such that for any u and v, the (unique) path from u to v can be represented by $O(\log n)$ paths and $O(\log n)$ other edges.

After you learn the technique, you can test your understanding at `http://www.spoj.com/problems/GSS7/`.

Heavy-light Decomposition

Pick some vertex r as the root of the tree. Define $s(u)$ to be the number of vertices in the subtree rooted at u, for $u \in T$. Define the set of children of u to be $C(u)$. For each vertex u, for $v \in C(u)$, if v is the one with the smallest index that satisfies $s(v) = \max_{w \in C(u)} s(w)$, then (u, v) is defined to be a *heavy* edge; otherwise, we define (u, v) to be a *light* edge.

Note that the heavy edges form paths, since each heavy edge can only have at most one adjacent heavy edge on each of the end points. We define paths consisting of heavy edges only to be *heavy paths*.

Heavy-light Decomposition: Structural Theorem

Theorem

For any u and v in T, the unique path from u to v is covered by at most $O(\log n)$ heavy paths, and $O(\log n)$ light edges.

Proof.

Define a to be the vertex on the path from u to v that is closest to r. We consider the path from a to u, and the case from a to v is similar. Denote the path from a to u by P. Denote the number of light edges in P by L. Observe that the number of heavy paths is at most $|L| + 1$, so it is sufficient to prove $|L| \leq O(\log n)$.
Suppose $L = \{e_1, e_2, \ldots, e_l\}$, where e_i's are ordered along the path P. For each $e = (x, y) \in L$ (where x is closer to a), we have that $s(y) \leq \frac{1}{2}s(x)$, by the selection of the light edges. Therefore, if u' is some vertex in P after visiting $|L|$ light edges, then
$1 \leq s(u') \leq s(a) \cdot \frac{1}{2^{|L|}}$. Since $s(a) \leq n$, we know that
$|L| \leq O(\log n)$. □

Heavy-light Decomposition: Application

Assuming that we have built a heavy-light decomposition, by the structural theorem, we can update/query any path via update/query $O(\log n)$ heavy paths and $O(\log n)$ light edges. If we use Segment Trees to maintain the heavy paths, and brute force the light edges, we immediately have a $O(\log^2 n)$ algorithm, where one $\log n$ comes from the number of heavy paths, and another comes from the Segment Tree.

It remains to construct the heavy-light decomposition efficiently. In fact, we can construct it in linear time.

Heavy-light Decomposition: Construction

The heavy-light decomposition can be constructed by two DFS's.

We perform a DFS to calculate $s(u)$ for all $u \in T$, using recursion $s(u) = 1 + \sum_{v \in C(u)} s(v)$. In the same DFS, we can also label the edges to be either heavy or light.

We perform another DFS on the graph with heavy edges only, and label the connected components. Note that each connected component corresponds to a heavy path.

If we use the Segment Tree on heavy path, we can use linear time to initialize the Segment Tree on the heavy paths.

Heavy-light Decomposition: Summary

Using the heavy-light decomposition, we decompose the tree into paths, so that any path on the tree can be represented by $O(\log n)$ heavy paths, and $O(\log n)$ light edges.

Therefore, we can use techniques for chains to solve problems on the paths on trees.