Notes for Lecture 12

1 Analysis of Union-Find

(Replaces Section 2 in the handout of October 9)

Suppose we initialize the data structure with n `makeset` operations, so that we have n elements each forming a different set of size 1, and let us suppose we do a sequence of k operations of the type `union` or `find`. We want to get a bound on the total running time to perform the k operations. Each `union` performs two `find` and then does a constant amount of extra work. So it will be enough to get a bound on the running time needed to perform $m \leq 2k$ `find` operations.

Let us consider at how the data structure looks at the end of all the operations, and let us see what is the rank of each of the n elements. First, we have the following result.

Lemma 1

If an element has rank k, then it is the root of a subtree of size at least 2^k.

Proof: An element of rank 0 is the root of a subtree that contains at least itself (and so is of size at least 1). An element u can have rank $k+1$ only if, at some point, it had rank k and it was the root of a tree that was joined with another tree whose root had rank k. Then u became the root of the union of the two trees. Each tree, by inductive hypothesis was of size at least 2^k, and so now u is the root of a tree of size at least 2^{k+1}. □

Let us now group our n elements according to their final rank. We will have a group 0 that contains elements of rank 0 and 1, group 1 contains elements of rank 2, group 2 contains elements of rank in the range $\{3, 4\}$, group 3 contains elements of rank between 5 and 16, group 4 contains elements of rank between 17 and 2^{16} and so on. (In practice, of course, no element will belong to group 5 or higher.) Formally, each group contains elements of rank in the range $(k, 2^k]$, where k itself is a power of a power ... of a power of 2. We can see that these groups become sparser and sparser.

Lemma 2

No more than $n/2^k$ elements have rank in the range $(k, 2^k]$.

Proof: We have seen that if an element has rank r, then it is the root of a subtree of size at least 2^r. It follows that there cannot be more than $n/2^r$ elements of rank r. The total number of elements of rank between $k+1$ and 2^k is then at most

$$n \sum_{r=k+1}^{2^k} \frac{1}{2^r} < n \sum_{r=k+1}^{\infty} \frac{1}{2^r} = \frac{n}{2^k} \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{n}{2^k}$$

□

By definition, there are no more than $\log^* n$ groups.

To compute the running time of our m operations, we will use the following trick. We will assign to each element u a certain number of "tokens," where each token is worth $O(1)$ running time. We will give out a total of $n \log^* n$ tokens.

We will show that each `find` operation takes $O(\log^* n)$ time, plus some additional time that is paid for using the tokens of the vertices that are visited during the `find` operation. In the end, we will have used at most $O((m + n) \log^* n)$ time.

Let us define the token distribution. If an element u has (at the end of the m operations) rank in the range $(k, 2^k]$ then we will give (at the beginning) 2^k tokens to it.

Lemma 3
We are distributing a total of at most $n \log^* n$ tokens.
Proof: Consider the group of elements of rank in the range $(k, 2^k]$: we are giving 2^k tokens to them, and there are at most $n/2^k$ elements in the group, so we are giving a total of n tokens to that group. In total we have at most $\log^* n$ groups, and the lemma follows. □

We need one more observation to keep in mind.

Lemma 4
At any time, for every u that is not a root, $rank[u] < rank[p[u]]$.
Proof: After the initial series of `makeset`, this is an invariant that is maintained by each `find` and each `union` operation. □

We can now prove our main result

Theorem 5
Any sequence of operations involving m `find` operations can be completed in $O((m + n) \log^* n)$ time.

Proof: Apart from the work needed to perform `find`, each operation only requires constant time (for a total of $O(m)$ time). We now claim that each `find` takes $O(\log^* n)$ time, plus time that is paid for using tokens (and we also want to prove that we do not run out of tokens).

The accounting is done as follows: the running time of a `find` operation is a constant times the number of pointers that are followed until we get to the root. When we follow a pointer from u to v (where $v = p[u]$) we charge the cost to `find` if u and v belong to different groups, or if u is a root, or if u is a child of a root; and we charge the cost to u if u and v are in the same group (charging the cost to u means removing a token from u's allowance) . Since there are at most $\log^* n$ groups, we are charging only $O(\log^* n)$ work to `find`. How can we make sure we do not run out of coins? When `find` arrives at a node u and charges u, it will also happen that u will move up in the tree, and become a child of the root (while previously it was a grand-child or a farther descendent); in particular, u now points to a vertex whose rank is larger than the rank of the vertex it was pointing to before. Let k be such that u belongs to the range group $(k, 2^k]$, then u has 2^k coins at the beginning. At any time, u either points to itself (while it is a root) or to a vertex of higher rank. Each time u is charged by a `find` operation, u gets to point to a parent node of higher and higher rank. Then u cannot be charged more than 2^k time, because after that the parent of u will move to another group. □

2 Introduction to Dynamic Programming

Recall our first algorithm for computing the n-th Fibonacci number F_n; it just recursively applied the definition $F_n = F_{n-1} + F_{n-2}$, so that a function call to compute F_n resulted in two functions calls to compute F_{n-1} and F_{n-2}, and so on. The problem with this approach was that it was very expensive, because it ended up calling a function to compute F_j for each $j < n$ possibly very many times, even after F_j had already been computed. We improved this algorithm by building a table of values of Fibonacci numbers, computing F_n by looking up F_{n-1} and F_{n-2} in the table and simply adding them. This lowered the cost of computing F_n from exponential in n to just linear in n.

This worked because we could sort the problems of computing F_n simply by increasing n, and compute and store the Fibonacci numbers with small n before computing those with large n.

Dynamic programming uses exactly the same idea:

1. Express the solution to a problem in terms of solutions to smaller problems.

2. Solve all the smallest problems first and put their solutions in a table, then solve the next larger problems, putting their solutions into the table, solve and store the next larger problems, and so on, up to the problem one originally wanted to solve. Each problem should be easily solvable by looking up and combining solutions of smaller problems in the table.

For Fibonacci numbers, how to compute F_n in terms of smaller problems F_{n-1} and F_{n-2} was obvious. For more interesting problems, figuring out how to break big problems into smaller ones is the tricky part. Once this is done, the the rest of algorithm is usually straightforward to produce. We will illustrate by a sequence of examples, starting with "one-dimensional" problems that are most analogous to Fibonacci.

3 String Reconstruction

Suppose that all blanks and punctuation marks have been inadvertently removed from a text file, and its beginning was polluted with a few extraneous characters, so the file looks something like "lionceuponatimeinafarfarawayland..." You want to reconstruct the file using a dictionary.

This is a typical problem solved by dynamic programming. We must define what is an appropriate notion of *subproblem*. Subproblems must be ordered by *size*, and each subproblem must be easily solvable, once we have the solutions to all smaller subproblems. Once we have the right notion of a subproblem, we write the appropriate recursive equation expressing how a subproblem is solved based on solutions to smaller subproblems, and the program is then trivial to write. The complexity of the dynamic programming algorithm is precisely the total number of subproblems times the number of smaller subproblems we must examine in order to solve a subproblem.

In this and the next few examples, we do dynamic programming on a one-dimensional object—in this case a string, next a sequence of matrices, then a set of strings alphabetically ordered, etc. The basic observation is this: *A one-dimensional object of length n has about*

n^2 *sub-objects* (substrings, etc.), where a sub-object is defined to span the range from i to j, where $i, j \leq n$. In the present case a subproblem is to tell whether the substring of the file from character i to j is the concatenation of words from the dictionary. Concretely, let the file be $f[1 \ldots n]$, and consider a 2-D array of Boolean variables $T(i, j)$, where $T(i, j)$ is true if and only if the string $f[i \ldots j]$ is the concatenation of words from the dictionary. The recursive equation is this:

$$T(i,j) = dict(x[i \ldots j]) \vee \bigvee_{i \leq k < j} [T(i,k) \wedge T(k+1,j)]$$

In principle, we could write this equation verbatim as a recursive function and execute it. The problem is that there would be *exponentially many* recursive calls for each short string, and 3^n calls overall.

Dynamic programming can be seen as a technique of implementing such recursive programs, that have heavy overlap between the recursion trees of the two recursive calls, so that the recursive function is called once for each distinct argument; indeed the recursion is usually "unwound" and disappears altogether. This is done by modifying the recursive program so that, in place of each recursive call a table is consulted. To make sure the needed answer is in the table, we note that the lengths of the strings on the right hand side of the equation above are $k - i + 1$ and $j - k$, a both of which are shorter than the string on the left (of length $j - i + 1$). This means we can fill the table in *increasing order of string length.*

for $d := 0$ to $n - 1$ do ... $d + 1$ is the size (string length) of the subproblem being solved
    for $i := 1$ to $n - d$ do ... the start of the subproblem being solved
    $j = i + d$
    if $dict(x[i \ldots j])$ then $T(i,j)$:=true else
        for $k := i$ to $j - 1$ do
        if $T(i,k)$ =true and $T(k+1,j)$ =true then do $\{T(i,j)$:=true$\}$

The complexity of this program is $O(n^3)$: three nested loops, ranging each roughly over n values.

Unfortunately, this program just returns a meaningless Boolean, and does not tell us how to reconstruct the text. Here is how to reconstruct the text. Just expand the innermost loop (the last assignment statement) to

$\{T[i,j]$:=true, first$[i,j] := k$, exit for$\}$

where first is an array of pointers initialized to *nil*. Then if $T[i,j]$ is true, so that the substring from i to j is indeed a concatenation of dictionary words, then first$[i,j]$ points to the end of the first word. Notice that this improves the running time, by exiting the for loop after the first match; more optimizations are possible. This is typical of dynamic programming algorithms: Once the basic algorithm has been derived using dynamic programming, clever modifications that exploit the structure of the problem speed up its running time.