Complexity and Advanced Algorithms – Assignment 4

Siddharth Bhat (20161105)

October 4, 2018

1 Doubly logarithmic tree

We follow the defintion of a doubly logarithmic tree from JaJa. Let $n = 2^{2^k}$. A doubly logarithmic tree with n leaves is one where **each node** at the ith level has $2^{2^{k-i-1}}$ children for $0 \leq i \leq k-1$.

Each node at the penultimate level k is defined to have 2 children.

For example, if $k = 2$, $n = 2^{2^2} = 2^4 = 16$, and the number of children at each level will be:

$$\begin{aligned} i = 0 &\mapsto 2^{2^{2-0-1}} = 2^{2^1} = 2^2 = 4 \\ i = 1 &\mapsto 2^{2^{2-1-1}} = 2^{2^0} = 2^1 = 2 \\ i = 2 = k &\mapsto 2 \end{aligned}$$

1.1 Depth of $O(\log \log n)$

By definition, the tree has k levels. Since, $n = 2^{2^k}$, $k = \log(\log n)$.

1.2 Number of nodes at level i is $2^{2^k - 2^{k-i}}$

Let us denote number of nodes at level i as $nodes(i)$. First, notice that:

$$nodes(i) = nodes(i-1) \times (\text{number of children at level } i-1)$$

by definition of us having a tree structure.

Proof. We prove the given equality by induction on i, the level of the tree.

1.2.1 i = 0

When $i = 0$, we have 1 node. From the formula, $nodes(0) = 2^{2^k - 2^{k-i}} = 2^{2^k - 2^{k-0}} = 2^0 = 1$

1.2.2 i = k + 1

We assume that $nodes(i) = 2^{2^k - 2^{k-i}}$.

From the recurrence written above,

$$\begin{aligned} nodes(i+1) &= nodes(i) \times (\text{number of children at level } i) \\ &= nodes(i) \times (\text{number of children at level } i) \\ &= nodes(i) \times 2^{2^{k-i-1}} \\ &= 2^{2^k - 2^{k-i}} \times 2^{2^{k-i-1}} \\ &= 2^{2^k - 2^{k-i} + 2^{k-i-1}} \\ &= 2^{2^k - 2 \cdot 2^{k-i-1} + 2^{k-i-1}} \\ &= 2^{2^k - 2^{k-i-1}} \\ &= 2^{2^k - 2^{k-(i+1}} \end{aligned}$$

Hence, $nodes(i+1)$ is consistent with the definition, and is therefore proved.

□

2 Problem 2

- Target: Time: $O(\log n)$. Ops $O(n)$.
- A - solves problem. Time: $O(\log n)$. Ops: $O(n \log n)$. **A exceeds target in target Ops**.
- B - reduces size by a constant factor (say $\frac{1}{2}$). Time: $O(\log n / \log \log n)$. Ops: $O(n)$. **C exceeds target in target Time**.

2.1 Analysis

Notice that we cannot directly solve the problem by using A, since A takes $O(n \log n)$ operations.

The only other option available is to repeat B till the problem size becomes small enough that we can run A.

Assume we repeat B for k rounds. This will bring the problem size from n to $n' = n/k$. If we wish for this reduced problem to be solved by A, then this takes $O(n' \log n')$ operations. For our target operations constraint, we require that:

$O(n' \log n') = O(n)$
$O(n/k \log n/k) = O(n)$
This means that $\dfrac{\log n/k}{k} = O(1)$. Solving this:
$\dfrac{\log n/k}{k} = O(1)$
$\dfrac{\log n}{k} - \dfrac{\log k}{k} = O(1)$

The only solution for this is $k = \log n$.

Howveer, if $k = \log n$, then to repeat problem B for k rounds, we require $k \cdot O(n) = n \log n$ operations!

So, it appears to be unsolvable using the above mentioned strategy, to get precisely the time bounds requested.

2.2 Approximate Solution

```
def solve(P):
    n = size(P)

    # Reduce problem size of log (log n) rounds
    for _ in range(log(log(n))):
        P = B(P)

    # Solve problem of size n' with A.
    A(P)
```

We first repeat algorithm B for r rounds, where $r \equiv \log \log n$. This gives us $O(\log n/ \log \log n \times r) = O(\log n)$ time.

This uses operations $O(n \log \log n) \approx O(n)$. Here, we perform the approximation that $\log \log n \approx O(1)$, which strictly speaking is incorrect, but is practically correct.

Running B for r rounds reduces the problem size from n to $n/2^r = n/2^{\log \log n} = n/ \log n$. Let n' be the reduced problem size, where $n' = n/ \log n$.

Now, let's check that running A on a problem of size n' does not use too many ops (since this was the bottleneck with problem size n):

$n' \log n' = (n/ \log n) \log(n/ \log n) = n/ \log n(\log n - \log \log n) = n - n \log \log n/ \log n < O(n)$.

Hence, Problem A will finish in the stipulated time.

3 All nearest smallest values to merging arrays

We are given a solution of $ANSV$ which runs in time $O(t(n))$ and $W(n)$ work. We must use this to merge to arrays of size $n/2$ each.

Assume $n = 2k$ to make the analysis simpler. We must merge arrays of size $n/2 = k$ each.

We first assume that the two arrays A, B are disjoint. We will extend the analysis to the non-disjoint case later.

Define $rank(x, A) = |\{y \mid y \in A, y < x\}|$. That is, the rank of an element x in a set A is the number of elements less than x in A. Note that $sort(A)[rank(x, A)] = x$. That is, $rank(x, A)$ is the index of x if A were sorted.

Let S be a sorted array of length n. Let us create a new array S' which is S with an element e appended to it (that is, $S'[0..n-1] = S[0..n-1], S'[n] = e$). now, notice that:

Lemma 1. *Let S be a sorted array of length n and v be a value. Let $S' = S + [v]$. That is, S' is the array S with a new nth element of v.*

$ANSV(S', n){+}1 = ANSV$ of the nth element of S' = sorted position of v in S

Proof. $ANSV(S', n) = i$ means that $S'[i] < S'[n]$, and $\forall gt > i$, $S[gt] \not< S'[n]$, by definition.

However, since S is sorted, $S[gt] > S[i] \forall gt > i$, and $S[less] < S[i] \forall less < i$.

Hence,

$$S[0] < S[i] \ldots S[i] < k < S[i+1] \ldots S[n-1]$$

Hence, $rank(S, k) = |\{ix \in [0..n] \mid S[ix] < k\}| = |[0..i]| = i + 1 = ANSV(S', n) + 1$.

$\square$

So, merging A and B would be the same as finding $rank(x, A \cup B) = rank(x, A) + rank(x, B)$. It is to find $rank(x, A)$ that we will need to exploit the sorted structure of the two arrays, and the $ANSV$ function.

3.1 Algorithm 1

This is based on a modified algorithm that uses $ANSV$ to find the rank of an element, instead of binary search.

```python
# implemented for us
def ANSV(sorted_arr, index):
    """Time complexity of O(t(n))"""
    pass


def rank(elem, sorted_arr):
    """Find the rank of element elem in a *sorted* array sorted_arr.
       Time complexity of O(t(n))
    """
```

```
    return ANSV(sorted_arr + [elem], length(sorted_arr)) + 1

def merge(A, B):
    """ Merges two arrays A, B of length n / 2.
        Time complexity of O(t(n))
    """

    # Results stored in `out` array of length `n`

    for 1 <= i <= n / 2 pardo:
        # the rank of B[i] in B is i
        # we can find the rank of B[i] in A by using rank()
         # Time: O(t(n))
         bi_rank = i + rank(B[i], A)

         # Time: O(1)
         out[bi_rank] = B[i]

    return out
```

3.1.1 Work & Time complxity

we use n processors and $t(n) + O(1) = t(n)$ time complexity, since we make n parallel calls to `rank`.

This makes makes the work $W = n \times t(n)$.