Selected Activity Solutions

Suhas Arehalli

COMP221 - Spring 2024

8: Sorting as Pre-Processing

1. **Sorting for Search**

   First observe that if we make a constant number of searches, sorting is always worse. How do we observe this?

   Well, if we want to be very formal about it, let $f(n) \in \Theta(n \log n)$ be a worst-case growth function for our sort, let $g(n) \in \Theta(n)$ be a worst-case growth function for our linear search, and let $h(n) \in Theta(\log n)$ be a worst-case growth function for our binary search implementation.

   Let $t(n)$ be the number of searches we do in terms of n. if we do a constant number of searches ($t(n) \in \Theta(1)$), then $t(n) = k \in \mathbb{Z}$. Then our total cost without sorting is

   $$t(n)g(n) = kg(n) \in \Theta(n)$$

   A constant amount of linear time searches is linear time. Makes sense! but with sorting

   $$f(n) + t(n)h(n) = f(n) + kh(n) \in \Theta(n \log n)$$

   Here, our sort dominates! $t(n)h(n) \in \Theta(\log n)$, which is great, but $f(n) \in Theta(n \log n)$, and when we have sums of time complexities, the dominant term determines our final time complexity.

   When does the sort stop dominating the time complexity? Well, if $t(n) \in \Theta(n)$, then $t(n)h(n) \in \Theta(n \log n)$, matching the time complexity of the sort! Meanwhile, the no-sorting option gives us $t(n)g(n)$, the product of two $\Theta(n)$ functions, which is $\Theta(n^2)$! So, with $t(n) \in \Theta(n)$, sorting pre-processing wins!

   We can then see if we can find the transition point. What if $t(n) \in \Theta(\log n)$? Not sorting gives us $\Theta(n \log n)$, while sorting gives us $f(n) \in \Theta(n \log n)$ and $t(n)h(n) \in \Theta(\log^2 n)$ —- a weird one, but certainly dominated by the $n \log n$ sort! This means that with $t(n) \in \Theta(\log n)$, both strategies are, in big-Θ terms at least, even!

   So our general advice would be: If we're searching more than roughly $\Theta(\log n)$ times, sorting is worth it!

2. **Good-Enough Subset Sum**

   Our strategy is sorting, and summing elements from left to right until we exceed k. This greedy strategy turns out to work! We haven't yet seem the proof strategy required (contradiction!), but here I'll briefly summarize the argument:

We find the maximum l such that the $l+1$ smallest elements sum to greater than k (you should write pseudocode and prove this about the code!). Call these elements, in sorted order $b_1 \dots b_{l+1}$, and note that this means that $\sum_{1 \le i \le l} b_i \le k$, but $\sum_{1 \le i \le l+1} b_i > k$.

Suppose that this were the wrong answer. Then there exist elements $a_1, \dots a_{l+1} \in A$ such that $\sum_i a_i \le k$ that some dastardly enemy of mine might have chosen to prove me wrong. In that case, I considered the $l+1$ *smallest* elements, so for each a_i, I can pair it with one of my elements, call them b_i, and find that $a_i \ge b_i$. That is, I can take their smallest element a_i and my smallest b_i, and either we picked the same elements (they're equal) or they picked a different one (and mine is smaller!), repeating this for the second smallest, and so on. If we let $a_1, \dots a_{l+1}$ and $b_1 \dots b_l$ be in sorted order (which won't change anything!), we have

$$a_i \le b_i, \forall 1 \le i \le l+1$$
$$\sum_{1 \le i \le l+1} a_i \le \sum_{1 \le i \le l+1} b_i$$

But this creates a contradiction, because the smaller size, $\sum_{1 \le i \le l+1} a_i$, was given to be *larger* than k, while our dastardly enemy's sum $\sum_{1 \le i \le l+1} b_i$ was supposed to be *less than or equal to k*. Thus, we conclude that no such $a_1 \dots a_{l+1}$ exists, and thus our solution is correct!

This consists of a sort plus a linear search, and so the time complexity of the sort (for us, as of now, $\Theta(n \log n)$) dominates.

3. **Maximum Gap**:

   Our strategy: Sort, and then compare adjacent elements! The way we define the maximum gap is equivalent to asking what the maximum difference between adjacent elements in a sorted array is. Again, the $\Theta(n \log n)$ sorting dominates the linear search for the max difference.

   The LeetCode $\Theta(n)$ strategies are actually not *really* any more sophisticated, but use a clever mix of linear time sorting techniques (i.e., bucket sort) as well as a tactic from divide and conquer algorithms (similar to the Maximum Subrange or Closest Points algorithms we'll see later!) to make our linear time sort a bit more general for this specific use case (linear sorts usually make strong assumptions about the kinds of items they can sort)!

10: Binary Search Variants

1. **Approximately How Wrong**

   First, we should observe that we begin with an interval of length $r-l$, and we're guaranteed that some root x lies in this interval. The bisection method has us construct the midpoint $m = l + \frac{r-l}{2}$ and determine either $l \le x \le m$ or $m \le x \le r$. Note that either way, the size of our interval is now $\frac{r-l}{2}$!

   This gives us our loop invariant: After k iterations of our loop, the width of the interval we know a root to be in becomes $\frac{r-l}{2^k}$.

   Now there's just one trick left. We want to solve for k such that we can pick x' such that $x - \varepsilon \le x' \le x + \varepsilon$. If we pick the midpoint of the interval we find to be x', we can see that our interval needs to be 2ε wide. So we solve for k:

$$\begin{aligned}\frac{r-l}{2^k} &= 2\varepsilon \\ r-l &= 2^{k+1}\varepsilon \\ 2^{k+1} &= \frac{r-l}{\varepsilon} \\ k+1 &= \log(r-l) - \log\varepsilon \\ k &= \log(r-l) - \log\varepsilon - 1\end{aligned}$$

Of course, if we're being very careful, we note that k needs to be an integer (no fractional loop!), so in actuality, we need $k \geq \log(r-l) - \log\varepsilon - 1$.

2. **With great power...**

   My actual solution is described in the section titled "Determining Power" in the linked paper. In short, perform a one-sided binary search to find the first n such that you have $> .8$ power, and then run regular binary search for some fixed number of iterations and pick the smallest n such that you get $> .8$ power.

11: The Master Theorem

Here checking your work entails comparing your intuitions to the master theorem! I'll work out how I think through 2.

$$T(n) = 3T(\frac{n}{2}) + n$$

Observe that here, $a = 3$, $b = 2$. This means that Master Theorem compares $f(n)$ to $\Theta(n^{\log_2 3})$. Well, $log_2 3 > 1$, and $f(n) \in \Theta(n)$, so $f(n) \in O(n^c)$ where $c < log_a b$. The Master Theorem tells us that, in this case $T(n) \in \Theta(n^{\log_2 3})$.

This means that the leaves dominate! Remember that $n^{log_b a} = a^{log_b n}$, which means that we have $n^{\log 23}$ leaves in our recursion tree, which means $n^{\log_2 3}$ base cases that each take $\Theta 1$ time to complete ($= \Theta(n^{\log 23})$ work just on the leaves!).

Now let's draw out the levels of our recursion tree to confirm our intuitions.

Level 0 will take $f(n) = n$ steps (this is dominated by $\Theta(n^{23})$ because $\log 23 > 1$!). The next level will create 3 problems of size $\frac{n}{2}$, which will take $\frac{3n}{2}$ time steps total. The second (0-indexed!) level will have 9 (3 children from 3 nodes on the second level!) problems of size $\frac{n}{4}$ (half the problem size of the previous level. This will take $\frac{9n}{4}$ work. We can start to see how every levels work is going to be dominated by the work done at the leaves! But we can and should be more careful to check our intuitions!

We can now identify a pattern: Level k will take $(\frac{3}{2})^k n$ work. So the problem, in total, will take

$$\sum_{k=0}^{d} (\frac{3}{2})^k n$$

time steps, where d is the depth of the tree! How deep is the tree? Well, if problems get halved each time, this gets us a depth of approximately $\lceil \log_2 n \rceil$. For clarity, I'll just write $\log_2 n$, since this essentially a big-Θ bound on $\lceil \log 2n \rceil$. Now we just plug this into a geometric series formula:

$$\begin{aligned}
\sum_{k=0}^{\log_2 n} (\frac{3}{2})^k n &= n\frac{1-(3/2)^{log_2 n+1}}{1-(3/2)} \\
&= -2n(1-(3/2)(3/2)^{log_2 n}) \\
&= -2n(1-(\frac{3}{2})\frac{3^{log_2 n}}{2^{log_2 n}}) \\
&= -2n(1-(\frac{3}{2})\frac{n^{log_2 3}}{n}) \\
&= -2n + 2(\frac{3}{2})n^{log_2 3} \\
&= 3n^{log_2 3} - 2n
\end{aligned}$$

Wild! We end up with something that's $\Theta(n^{\log_2 3})$, just as the Master Theorem says!

12: More Divide & Conquer Design

1. To solve the first problem with a Divide & Conquer strategy, first consider the hint: If I rob house i, then I get $A[i]$ dollars, but can't rob houses $i-1$ or $i+1$ lest I tempt the authorities. I could also *not* rob house i, and get the opportunity to rob houses $i-1$ and $i+1$, but not earn $A[i]$ dollars from house i.

   The first trick here is to realize that there are downstream effects: if I don't rob house i, but then decide to rob house $i+1$, I can't rob house $i+2$, and so on all the way along. Another way to see it is that picking one house pits maximizing earnings from a particular house against maximizing the number of houses I rob! If I rob an odd house, I can either rob *all* odd houses to get half the houses, or I can choose to at some point *skip* a house and swap over to even houses. But, since you skipped a house, you're robbing one less house than if you only robbed odd or even houses!

   The idea is that this is messy, downstream thinking. The second trick is to realize that this is *recursive* thinking. If I rob house i, I can't rob $i-1$ or $i+1$. But I can rob houses represented by $A[1 \dots i-2]$ and $A[i+2 \dots N]$. How much do I make if I rob house i, well, the maximum I can rob from $A[1 \dots i-2]$ plus the maximimum I can rob from $A[i+2 \dots N]$ plus $A[i]$! And observe that asking how much I can rob from $A[1 \dots i-2]$ looks exactly like the question we're asking about A as a whole! Thus, if we call our function $MaxEarnings$ we can express how much we earn from robbing house i as

   $$e_{rob} = MaxEarnings(A[1 \dots i-2]) + A[i] + MaxEarnings(A[i+2 \dots N])$$

   But this only consider if we rob house i. We could leave it alone! How much can we make here? Well, 0 from house i, but every other house is open! So

   $$e_{don't} = MaxEarnings(A[1 \dots i-1]) + MaxEarnings(A[i+1 \dots N])$$

And so we can compute how much we can earn if we rob i, e_{rob} and how much we can earn if we don't, $e_{don't}$. Well, the maximum we can earn is then $\max(e_{rob}, e_{don't})$!

One more question: We haven't specified i yet! For the purposes of our analysis, let's choose $i = \frac{N}{2}$. This will make applying the master theorem easy, but functionally, it won't matter.

```
function MaxEarnings($A$)
    $m \leftarrow \frac{N}{2}$
    $e_{rob} \leftarrow A[i]$ + MaxEarnings(A[1...m-2]) + MaxEarnings($A[m+2 \dots N]$)
    $e_{don't} \leftarrow$ MaxEarnings(A[1...m-1]) + MaxEarnings($A[m+1 \dots N]$)
    return $\max(e_{rob}, e_{don't})$
end function
```

Now to get a time complexity, note that we make 4 recursive calls, each on a problem of approximately size $\frac{N}{2}$, and outside of these recursive calls, we do $\Theta(1)$ work (just basic math ops!). This means the Master Theorem compares $\Theta(1)$ work at each level to $n^{\log 24} = n^2$ leaves, and we find that the leaves dominate and our runtime is $\Theta(n^2)$!

Note that a better solution exists (Dynamic Programming, soon!), but this is still **much** better than a brute force solution ($\Theta(2^n)$ time!).

2. This one is binary search with a twist!

   Let's analyze what happens when we just pick some i.

Case 1: $A[i] = i$. Well, we're done! Nothing to see here!

Case 2: $A[i] > i$. Now we have to look at my carefully highlighted words and the hint! if we look at the next index, it must be *at least* 1 greater (no duplicates, integers!). so $A[i+1] \geq A[i] + 1$. But we know $A[i] > i$, so...

$$A[i+1] \geq A[i] + 1 > i + 1$$

Oh no, the next element will be bigger than it's index too! We can construct a quick inductive argument from this!

Statement: $\forall k > i$, $A[k] > k$

Base Case: For $k = i$, $A[k] > k$

Inductive Step: Assume $A[k] > k$. We must show $A[k+1] > k+1$. We simply observe as above that $A[k+1] \geq A[k] + 1$, but since by our IH we have that $A[k] > k$, $A[k+1] \geq A[k] + 1 > k + 1$, as desired.

Note that this means that the only indices that can have $A[k] = k$ are those with $k < i$, so we recurse on $A[1 \dots i-1]$!

Case 3: $A[i] < i$. Similarly, we can construct a parallel inductive argument that if $A[k] < k$, $A[k-1] \leq A[k] - 1 < k - 1$, meaning that for all $k < i$, $A[k] < k$. This means that we recurse on $A[i+1 \dots N]$.

Note that if we choose $i = \frac{N}{2}$, this cuts our array size in half for a single recursive call. This is our binary search-like strategy! The master theorem will confirm that this runs in $\log n$ time (a=1, b=2, $\Theta(1)$ work outside of a recursive call).

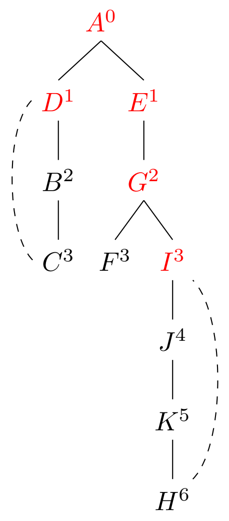

Figure 1: Annotated DFS Tree. Superscripts are depths.

13: Graph Traversal

1. **Articulate a Plan** *Note that I've almost entirely re-written this activity because the original version was far too vague about what I wanted you to do.*

   Fig. 1 is an annotated DFS tree from A. Here, we note that our cut-vertices are A, E, G, I, and D.

   There are 3 things this problem wants you to consider:

   (a) First, that an internal node is a cut vertex only if some descendent of that node can't form a path through a backward edge back up to the rest of the DFS tree.

   (b) Second, that depths are sufficient to compute this: Backward edges in a DFS tree *must* go from a node to an ancestor of that node, so if a node's children can reach backward to a depth lower than that of a parent, removing that parent won't disconnect the graph.

   (c) Third, that merely looking at the children is insufficient (you can probably gather this from Skiena's version!). Consider a node like J. It's child, K, can't use a backward edge to get to I or higher. However, it's child H does have a backward edge that can get above J to a depth 3 vertex! This means that our solution can't just be local — it needs to be get information from lower in the tree!

   This points us to a DFS based algorithm (assuming a connected G) in Alg. 1.

2. **Covering for a Friend** Here, the solution depends on you recalling a core property of DFS trees on undirected graphs: The graph's edges are partitioned into *tree edges* and *backward edges* (and edge from v to an ancestor of v. This means that there **cannot** be cross edges (that connect a vertex v to a vertex w that is not an ancestor or descendent of v).

   Here, we simply have to proceed by contradiction: Assume for contradiction that the non-leaf vertices in the DFS tree do not form a vertex cover. That means that there exists some edge

Algorithm 1 An algorithm to find cut-vertices/articulation vertices in a connected undirected graph G. the idea is to mark depths on vertices, and return the highest reachable point from the vertex the recursive function was called on. We use an unspecified function MarkCutNode to mark the cut nodes we identify. The structure is DFS, plus the relevant depth/high checks and updates. Double check to make sure it handles leaves + root as well!

```
function CutVertex($G$)
    Select $s \in G$ at random.
    CutVertexRecursive($G$, $s$, 0)
end function
function CutVertexRecursive($G$, $v$, $d$)
    MarkDepth($v$, $d$)
    $high \leftarrow d$
    for $(v, v') \in E$ do
        if $v'$ is undiscovered then
            Discover($v'$)
            $d' \leftarrow$ CutVertexRecursive($G, v', d+1$)
            $high \leftarrow \min(high, d')$
            if $d' \geq d$ then
                MarkCutNode($v$)
            end if
        else
            if Depth($v'$)$< d - 1$ then
                $high \leftarrow \min(high, \text{Depth}(v'))$
            end if
        end if
    end for
    return $high$
end function
```

$(v, w) \in E$ that is not incident to any non-leaf node! This implies that both v and w are leaves of the DFS tree. Of course, this means that $(v, w) \in E$ is a cross edge, since two leaves cannot be in an ancestor/descendent relationship (neither, by definition, can have children!)! This is a contradiction, since a DFS tree guarantees only backward edges! Thus, the non-leaf nodes in a DFS tree must be a vertex cover.

A more sophisticated extension of this would be proving that DFS trees can only contain tree and backward edges. The proof of this is not complicated — the core idea is that if we encountered such an edge (v, w) at a vertex v, we should either follow the edge now (making it a tree edge!) or have followed that edge earlier (also making it a tree edge!) — but it is tricky to get the book-keeping right to make the proof clean. It's useful to keep track of the *timing* of each visit by using in-degree and out-degree labels for vertices (how many steps before we discover the vertex? How about until we finish processing all of the recursive calls that come from this vertex?). This makes it a good exercise for you to do! It'll seem overwhelming at first, but the difficulty is just in juggling the numbers correctly!