Design and Analysis of Algorithms, Honors

Raymond Bian

February 1, 2024

Contents

Lecture 1: Intro and Review

1 Intro to Algorithms

1.1 Properties of Algorithms

What kind of properties can an algorithm even have?

Property. Algorithms can be categorized by speed, memory, readability (simple to understand), accuracy (approximation quality), and requirements for the input.

The focus of this class will be on **speed**. We will also talk about accuracy, and algorithms will be mostly readable (but not always). We need some language that will allow us to describe algorithms.

1.2 Describing Algorithms

> **Example.** Let's take for example selection sort.

> **Explanation.** Using plain english, selection sort is: reapeatedly finding the smallest element and appending it to the output.

Algorithms can be described at different levels of detail. One extreme is very informally, like the sentence above. This type helps convey the **key concepts**, but can be rather vague, ambiguous, and hard to understand.

The other extreme could be posting the source code of the program. This would be a complete description of an algorithm, because even the computer and execute it. However, source code includes details that **only the computer needs**, such as types, etc.

So, when talking about algorithms, it makes sense to use a middleground, known as **pseudocode**.

> **Definition 1.** **Pseudocode** is a descriptive, step by step set of instructions that detail the structure of a program. English is allowed, and the level of detail depends on the context and the audience.

Algorithm 1 Pseudocode for Selection Sort

Any array of elements *input*
A sorted array *output*
1: **for** $i \leftarrow 1 \dots n$ **do**
2: Find $j \geq i$ with smallest $arr[j]$
3: Swap $arr[i]$ and $arr[j]$
4: **end for**

More detailed descriptions can also be given with pseudocode.

1.3 Runtime of Algorithms

Many factors can impact the runtime of an algorithm. For example, Selection Sort, when ran on the professor's laptop had a runtime of $0.017n^2 + 0.669n + 0.114$ms. This runtime will vary depending on the context, things like hardware, the programming language, and temperature.

Because we cannot determine these constants for every computer, we will just **ignore them** for analyzing algorithms.

> **Definition 2.** **Big-O Notation** is the notation used to analyze runtimes. It essentially ignores unknown constant factors. More formally, it states that $f(n) = O(g(n))$ if and only if there exists n_0 and $c > 0$ such that $f(n) \leq c \cdot g(n)\ \forall n \geq n_0$.

Lemma 1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = O(g(n))$.

We can find the runtime of algorithms by counting the number of operations.

Example. Selection sort runs in $O(n^2)$.

Explanation. We have

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = \frac{n(n-1)}{2} = O(n^2).$$

Note that the second equality comes from taking the limit.

Definition 3. **Omega** notation denotes the relation $\geq$.

Lemma 2. $\lim_{n\to\infty} \frac{g(n)}{f(n)} \Rightarrow f(n) = \Omega(g(n))$.

Definition 4. **Little-O** notation denotes the strict relation $<$.

Lemma 3. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$.

Definition 5. **Little Omega** notation denotes the strict relation $>$.

Lemma 4. $\lim_{n\to\infty} \frac{g(n)}{f(n)} \Rightarrow f(n) = \omega(g(n))$.

Definition 6. **Theta** notation denotes the equality relation $=$.

Lemma 5. $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n)) \Rightarrow f(n) = \Theta(g(n))$.

Lecture 2: Divide and Conquer

2 Divide and Conquer

The main idea of divide and conquer is to split the problem into smaller subproblems, and solve those subproblems by doing the same. Finally, we combine the solutions together to solve the main problem.

Example. One example of a divide and conquer is Merge Sort.

Algorithm 2 Merge Sort

A, **an array** *input*
Sorted array *output*

```
if n = 1 then
    return A
end if
left = MergeSort(A[0 ... ⌈n/2⌉ − 1])
right = MergeSort(A[⌈n/2⌉ ... n − 1])
return Merge(left, right)
```

Algorithm 3 Merge

L, R, **two sorted arrays** *input*
Merged sorted array *output*

```
l = 0, r = 0, i = 0
while l < a and r < b do
    if L[l] < R[r] then
        output[i] = L[l]
        l += 1
    else
        output[i] = R[r]
        r += 1
    end if
    i += 1
end while
if l = a then
    output.append(R[r ... b])
else
    output.append(L[l ... a])
end if
return state
```

However, how do we prove that such an algorithm is correct?

Lemma 6. Merge(L, R) correctly merges L and R such that the output is sorted if L and R are sorted.

Proof. We wish to show that after the i-th (1 indexed) iteration of the while loop, output$[1 \ldots i]$ is sorted. We will proceed with induction on i.

Base case: $i = 0$. In this case, nothing has occurred yet, and nothing is in the output, so the output is sorted.

Inductive step: By the inductive hypothesis, output$[0 \ldots i-1]$ is sorted. Let l, r be the value of variables l, r at the start of the i-th iteration. Note that output$[i-1]$ is

either

$$\begin{cases} L[l-1] & = \min(L[l-1], R[r]) \\ & \leq \min(L[l], R[r]) \\ & = \text{output}[i] \text{ if prev. was } L \\ R[r-1] & = \min(L[l], R[r-1]) \\ & \leq \min(L[l], R[r]) \\ & = \text{output}[i] \text{ if prev. was } R \end{cases},$$

which means that output$[i-1] \leq$ output[i], which means that output$[0 \dots i]$ is also sorted. After the while loop, output$[i-1]$ is either

$$\begin{cases} L[a] & \leq R[r] \\ & \Rightarrow \text{out.append}(R[r \dots b]) \text{ sorted} \\ R[b] & \leq L[l] \\ & \Rightarrow \text{out.append}(R[l \dots a]) \text{ sorted} \end{cases}.$$

Therefore, the output is a sorted array.

■

Theorem 1. Merge sort is correct.

Proof. We will process with induction on n.

Base case: $n = 1$. Then, the array A is just one element, such that A is sorted. MergeSort returns A, so it returns a sorted array.

Inductive step: We assume that merge sort works for all arrays with size $< n$. Let A be an array of length n. Note that $A[0 \dots \lceil \frac{n}{2} \rceil] - 1$ and $A[\lceil \frac{n}{2} \rceil \dots n-1]$ are both shorter arrays of length $< n$. Therefore, by the inductive hypothesis, we know these two subarrays must be sorted. Since merge is correct, the output must also be sorted.

■

Now, let us argue about the time complexity of merge sort.

Theorem 2. Merge sort has time complexity $n \log n$.

Proof. Let $T(n)$ be the time complexity for sorting an n-length array. Note that $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$. This is because merge is linear. We have the base case of $T(1) = 1$.

Let's look at the recursion tree. At the i-th level, there are 2^i calls to merge sort, and the input has length $\frac{n}{2^i}$. Each merge takes $O(\frac{n}{2^i})$ time, which implies that all merges performed at level i take $2^i \cdot O(\frac{n}{2^i}) = O(n)$ time.

Because we have $\log_2(n)$ levels, and each level takes $O(n)$ time, then merge sort takes $O(n \log n)$ time. ■

Theorem 3. In the general case, with a recursive calls and each subproblem of length $\frac{n}{b}$, and $O(n^c)$ time to combine solutions, we have $T(n) = a \cdot T(\frac{n}{b}) + O(n^c)$.

Proof. Then, the total time for level i is $O(a^i \cdot (\frac{n}{b^i})^c)$, and the total time is $O(\sum_{i=0}^{n^{\log_b(n)}} a^i (\frac{n}{b^i})^c)$, which has three cases.

$$\text{Total Time} = \begin{cases} O(n^{\log_b(a)}), & \text{if } a > b^c \\ O(n^c \log(n)), & \text{if } a = b^c \\ O(n^c), & \text{if } a < b^c \end{cases}.$$

■

Lecture 3: Divide and Conquer 2

One general approach that could be used for divide and conquer is, first, split the input into equal parts, assume you have solved the subparts, and then answer the question: how do we merge solutions? Note that sometimes we can improve the complexity of merge as well.

Example. Given an array $A[0 \dots n-1]$, find the max $A[i] - A[j]$ such that $i < j$.

Explanation. The naive iterative solution is iterating over all pairs i, j and checking the difference. Otherwise, we can use a divide and conquer solution given below.

```
def max_dif(A):
    if n = 1:
        return float('-inf')
    left = max_dif(A[:len(A)/2])
    right = max_dif(A[len(A)/2 + 1:])
    return max(left, right, max(left) - max
        (right))
```

Then, the complexity of this is $T(n) = 2 \cdot T(\frac{n}{2}) + O(n) = O(n \log n)$. Note that we can optimize this solution by optimizing the linear time merge to constant time. Instead of returning only the solution, we can return as well the maximum and minimum element of each of the arrays.

```
def max_dif(l, r, A):
    if l == r:
        return float('-inf'), A[l], A[l]
    m = (l + r) // 2
    left, min_l, max_l = max_dif(A[l:m])
```

```
    right, min_r, max_r = max_dif(A[m + 1:r
        ])
    return max(left, right, max_l - min_r),
         min(min_l, min_r), max(max_l,
        max_r)
```

2.1 Karatsuba Multiplication

Multiplication is not actually $O(n)$ - it is $O(n^2)$. This is because addition takes a lot longer as numbers n grow bigger. This is the algorithm used by the python interpreter.

Consider a number n bits long. Then, we can write $A = 2^{\frac{n}{2}} \cdot A_1 + A_2$ and $B = 2^{\frac{n}{2}} \cdot B_1 + B_2$. Then,

$$A \cdot B = 2^n A_1 \cdot B_1 + 2^{\frac{n}{2}}(A_1 B_2 + B_1 A_2) + A_2 B_2.$$

This multiplication only contains 4 multiplications of length (bits) $\frac{n}{2}$, which is faster (?). Using the Master Theorem to calculate, we only get an algorithm of $O(n^2)$ time.

We can continue to optimize by instead multiplying with the fact that

$$(A_1+A_2)\cdot(B_1+B_2) = A_1B_2+A_1B_2+B_1A_2+A_2B_2.$$

Let x be this value. Let $y = A_1 \cdot B_1$ and $z = A_2 \cdot B_2$. Then, we can instead return

$$2^n y + 2^{\frac{n}{2}}(x - y - z) + z.$$

Then, by the master theorem with merge (addition and subtraction) of $O(n)$, this algorithm has a runtime of $O(n^{\log_2(3)})$.

Lecture 4: Divide and Conquer 3

Note that arithmetic will take O(1) unless numbers blow up (grow exponentially).

Today we will cover a different version of divide and conquer. One example of this version is binary search.

```
def bin_search(arr, k):
    if n == 1:
        if arr[0] == k:
            return arr[0]
        else:
            raise Exception("")
        if a[n/2] < k:
            return bin_search(arr[n/2+1:],
                k)
        else:
            return bin_search(arr[:n/2], k)
```

By the master theorem, we have $a = 1, b = 2, c = 0$ such that binary search runs in $O(\log n)$ time. Another similar algorithm to binary search in which we only recurse into one spot is finding the k-th largest element in array, with a quickselect-type algorithm.

```
def k_smallest(arr, k):
    p = A[rand()]
    smaller = []
    larger = []
    for i in range(n):
        if arr[i] <= p:
            smaller.append(arr[i])
        else:
            larger.append(arr[i])
    if smaller = k:
        return p
    if smaller < k:
        return k_smallest(larger, k - len(
            smaller))
    if smaller > k:
        return k_smallest(smaller, k)
```

If we can guarantee a good pivot $\frac{3}{10}n \leq \text{smaller} \leq \frac{7}{10}n$, then the number of recursions is at most $\log_{\frac{10}{7}}(n) = O(\log n)$. We can guarantee these goods pivots with the median of medians algorithm.

First, we split our array A into $\frac{n}{5}$ groups, each of size 5. We can then sort each of these elements in "constant" time. Then, the median of each group is in the middle index of each 5. Now, we return the median of the array containing all medians. We can do this with recursion (divide and conquer), by calling ksmallest on it again.

We claim that the median of medians is a good p. Imagine the upper left rectangular part of the median array. This is less than or equal to the median of medians. Same with the lower right rectangular part. Note that the number of elements in the rectangular parts are both $\frac{3}{10}$. Therefore, p is at least the $\frac{3}{10}$-th element (same with the upper side).

Note that the median of medians algorithm takes constant time because sorting each of the 5 arrays is constant, so the sorting is linear. Then, selecting the median of the medians is also linear time, by a recursive call. We have a runtime of

$$T(n) + T\left(\frac{n}{5}\right) + O(n) + T\left(\frac{7}{10}n\right) = O(n).$$

Note that somehow, if we split into 3 groups, then the runtime is $n \log n$. However, splitting into any odd number greater than 3 works.

Lecture 5: Review for Exam 1

Theorem 4. Median of medians of $O(n)$.

Proof. We wish to show that

$$T(n) = T\left(\frac{n}{5}\right) + O(n) + T\left(\frac{7}{10}n\right) \leq C \cdot n.$$

We proceed by induction. We know that

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right) + d \cdot n.$$

by definition of $O(n)$.

Assume $T(n') \leq C \cdot n'$ for all $n' < n$. Therefore,

$$T(n) \leq C \cdot \frac{n}{5} + C \cdot \frac{7n}{10} + d \cdot n.$$

Simplifying, we have

$$T(n) \leq C \cdot n \cdot \left(\frac{2}{10} + \frac{7}{10}\right) + d \cdot n.$$

We pick $C = d \cdot 10$ such that

$$T(n) \leq C \cdot n.$$

as desired. Note that for the trivial base case, we know that this algorithm for $n = 1$ takes constant time. ■

Lecture 6: Graphs, DFS, Topological Order

3 Graphs

Insert every class' graph definitions here.

Two common representations of graphs are adjacency matrices, adjacency lists, and edge lists.

Definition 7. An **adjacency matrix** is a matrix where

$$A_{ij} = \begin{cases} 1 & \text{if there is an edge between } i, j \\ 0 & \text{otherwise} \end{cases}.$$

However, an adjacency list uses too much space, so an adjacency list can be used, which is at most $O(V + E)$ space. This list stores only direct neighbors.

3.1 DFS

Yeah, it's just DFS. At a current node, go to the next node. After you've finished exploring everything from that node, come back to this node and go to next node after. Keep a visited array to not visit vertices more than once.

Lemma 7. DFS(v) will visit every vertex that can be reached from v via some path.

Proof. By contradiction. Assume that there is a path $v, v_1, v_2, \ldots, v_k$. Assume v_k is not visited by the call to DFS(v). Let v_i be the last vertex that did get visited. That means DFS(v_i) must have been called. However, that means the edge between v_i and v_{i+1} must have been looked at. The only way then that DFS(v_{i+1}) doesn't get called, is if it is already visited. However, that means that DFS(v_{i+1}) must have been called. This is a contradiction, as then v_{i+1} must be the last visited vertex. ■

Definition 8. A graph G is a **directed acyclic graph** if it is a directed graph without cycles.

You can think of directed acyclic graphs as dependency graphs. What happens if we have a cycle? Bad things. How can we find these cycles? We can run DFS, and if we explore a vertex that is already in the current path, then we know there is a cycle. We can use backtracking to extract this cycle, if necessary.

3.2 Topological Sort

We can order the vertices of our graph topologically, so that all dependencies are processed before the nodes that require them. This is just a dfs where we only recurse once all vertex indegrees have been processed already.

Lecture 7: BFS, Dijkstra

3.3 BFS

Note that BFS is just DFS, but instead of processing the most recent node explored, we process the latest node explored. To find shortest paths, we can track the parent of each node, and go backwards.

Theorem 5. BFS finds the shortest path.

Proof. Let v_i be the vertices with distance i to the start point of BFS. We wish to show that BFS visits all vertices from v_i before visiting a vertex from v_{i+1}. We proceed by induction on i.

Base case: $i = 0$. Then, v_0 is just the start point which is visited first.

Step case: $i \to i + 1$. Let's look at BFS right after we finished visiting all vertices v_i. Note that no vertices of distance v_{i+1} have been visited, which implies that no vertices of distance v_{i+2} has been enqueued, and all vertices from v_{i+1} are currently in the queue. Since we use a FIFO queue all of v_{i+1} must be visited before a vertex from v_{i+2}. Furthermore, the edge we use to traverse to v_{i+1} requires that an edge (u, v_{i+1}) such that $u \in v_i$.

■

However, what happens if we want to find the shortest path in a graph where there are edge weights?

3.4 Dijkstra

Instead of a first in first out queue, we use a priority queue. This allows us to process vertices in a way such that the cost to travel to that vertex is less.

Theorem 6. Dijkstra is optimal.

Proof. Let $d(v)$ be the length of the shortest path from s to v for all $v \in V$, where s is the start vertex. Assume the algorithm stores an incorrect value in the distance array. Let $v_1, v_2, \ldots v_n$ be the vertices in the order in which they were visited by Dijkstra. Let v_i be the first vertex where $d(v_i)$ is wrong. That means we used the wrong edge to go from v_{i-1} to v_i. This means that

$$d(v_i) = d(v_{i-1}) + w_{v_{i-1}v_i} > \text{shortest path to } v_i.$$

Let a be a vertex with guaranteed correct distance, and b one otherwise. Assume that this path with a, b is the correct shortest path from s to v_i. However, v_i was popped from the queue before b. This means that the shortest path to b is $\geq$ the shortest path to v_i. With some algebraic manipulation as well as asserting that there are no negative edge weights (check the lecture notes), then we have that the shortest path to v_i is strictly greater than the shortest path to v_i, a contradiction. Therefore, there cannot exist such a v_i. ■