Design and Analysis of Algorithms, Honors

Raymond Bian

January 11, 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.

Proof. 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*
for $i \leftarrow 1 \dots n$ **do**
  Find $j \geq i$ with smallest $arr[j]$
  Swap $arr[i]$ and $arr[j]$
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 $O(n^2)$.

Proof. 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. $\lim\limits_{n\to\infty} \frac{g(n)}{f(n)} \Rightarrow f(n) = \Omega(g(n))$.

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

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

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

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

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.

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.

Algorithm 2 Merge Sort

A, **an array** *input*
Sorted array *output*
if $n = 1$ **then**
    return A
end if
left = MergeSort($A[0 \ldots \lceil \frac{n}{2} \rceil] - 1)$
right = MergeSort($A[\lceil \frac{n}{2} \rceil \ldots n-1]$)
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($R[r \ldots b]$)
else
    output.append($L[l \ldots a]$)
end if
return *state*

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$[0 \ldots i]$ is also sorted. After the while loop, output$[i-1]$ is either

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

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 $A[0 \ldots \lceil \frac{n}{2} \rceil] - 1$ and $A[\lceil \frac{n}{2} \rceil \ldots n-1]$ 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 $O(a^i \cdot \left(\frac{n}{b^i}\right)^c)$, and the total time is $O(\sum_{i=0}^{n^{\log_b(n)}} a^i \left(\frac{n}{b^i}\right)^c)$, which has three cases. □