Contents

Module 1 Quiz Review

An overview of quizzable content from Module 1, as described by the course webpage.

Basic terms / concepts

- **Algorithm**: roughly, a prescribed set of steps performed on an input to produce output that satisfies some specification
  - Every algorithm should be detailed using:
    - One-line problem description
    - Inputs
    - Outputs
    - Assumptions

    - Strategy overview—1 or 2 sentences outlining basic strategy, including name of "method" you might use
    - Algorithm description—step-by-step process or pseudocode
- **Basic operation:** the most primitive, granular "step" that we count while analyzing an algorithm's runtime
- **Brute force algorithm:** Algorithm that generates outputs by exhaustively testing possible solutions
- **Worst-case:** Analysis of an algorithm under the "worst-case" scenario; i.e., the scenario that results in the worst possible runtime/space complexity/etc.
- **Average-case:** Analysis of an algorithm in the "average case", or, perhaps, the "representative case." Includes some assumption about what kind of input is "average."
- **Amortized runtime:** The average of worst-case runtimes over several (n) executions. (Compare to average-case, which is still a measure of runtime for a single execution.)
- **Best-case:** Analysis of an algorithm in the best possible scenario; i.e., the scenario that results in the best possible runtime/space complexity/etc.

Asymptotic analysis

Asymptotic growth rate: A measure of how some function grows with respect to some input. Typically, the function measures runtime, space, or something else related to the execution of an algorithm for a given input size.

Order class: Sets that contain functions, with their membership determined by how they grow with input size.

Big order classes

O(g(n)) (Big-O, asymptotic upper bound):

- $f(n) \in O(g(n)) \iff \exists c, n_o$ s.t. $f(n) \leq c * g(n) \forall n > n_0$.
- $f(n)$ is less than or equal to $g(n)$ times some constant c for all n greater than some n_0.
- Conceptually, this means that $f(n)$ "grows no faster" than $g(n)$.

Ω(g(n)) (Big-Ω, asymptotic lower bound):

- $f(n) \in \Omega(g(n)) \iff \exists c, n_o$ s.t. $f(n) \geq c * g(n) \forall n > n_0$.
- $f(n)$ is greater than or equal to $g(n)$ times some constant c for all n greater than some n_0.
- Conceptually, this means that $f(n)$ "grows no slower" than $g(n)$.

Θ(g(n)) (Big-Θ, asymptotic tight bound):

- $f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$.
- $f(n)$ is of order classes $O(g(n))$ and $\Omega(g(n))$. (The relevant definitions apply, but note that the constants c need not be the same.)

- Conceptually, this means that $f(n)$ grows "exactly as fast" as $g(n)$ does.

Little order classes

o(g(n)) (Little-o)

- $f(n) \in o(g(n)) \iff f(n)/g(n) \to 0$ as $n \to \infty$
- $f(n)$ grows STRICTLY slower than $g(n)$. The growth of $g(n)$ always dominates that of $f(n)$.

ω**(g(n))** (Little-ω)

- $f(n) \in \omega(g(n)) \iff g(n)/f(n) \to 0$ as $n \to \infty$
- $f(n)$ grows STRICTLY faster than $g(n)$. The growth of $f(n)$ always dominates that of $g(n)$.

(Little-θ can't exist!)

Lower-bounds, optimal algorithms

A **lower-bound** provides an optimal runtime for some category of algorithms. If an algorithm achieves this lower bound, we say the algorithm itself is *optimal*; no other development can improve on its runtime.

Insertion sort

Sorting algorithm that repeatedly selects an element and inserts it in the correct position, building a sorted subarray as it continues

Advantages:

- In-place; no memory overhead
- Stable

Python code

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        j = i
        while j > 0 and arr[j] < arr[j-1]:
            # swap A[j] and A[j-1]
            arr[j], arr[j-1] = arr[j-1], arr[j]
            # decrement j
            j -= 1
```

Runtime

The best-case runtime of insertion sort is $\Theta(n)$. When the array is sorted, the inner loop never has to execute.

The worst-case runtime of insertion sort is $\Theta(n^2)$. If the elements are sorted in reverse order, each of the n elements is tested against the elements before it.

Insertion sort: "best of its breed"

Despite its quadratic runtime, insertion sort is actually an optimal algorithm that sorts by only swapping adjacent elements. Why is this?

Call an "inversion" any time when an array element is misordered relative to another element. In the maximal case, an array of n elements in reverse order will have $n + (n-1) + (n-2) + ... = \frac{n(n-1)}{2}$ inversions.

A single swap of adjacent elements can only resolve one inversion. So, in the worst case, a sorting algorithm has to swap $n(n-1)/2$ times. This corresponds with a $\Theta(n^2)$. An adjacenct swapping sorting algorithm will never be $o(n^2)$.

Divide-and-conquer

Class of recursive algorithms that follow three basic steps:

1. **Divide:** divide input problem into several subproblems.
2. **Conquer:** recursively solve each subproblem.
3. **Combine:** combine subproblem solutions into one big solution.

Mergesort: textbook D&C

Mergesort is a great example of a divide-and-conquer algorithm:

1. **Divide:** Divide input array into two subarrays.
2. **Conquer:** Recursively call MergeSort on each subarray.
3. **Combine:** Call `merge()` to combine the two individually sorted subarrays.

In efficient implementations, `merge` has linear runtime complexity.

Properties: - Stable (provided the right implementation of merge) - In-place

Mergesort runtime Assuming a linear merge, the recurrence relation for worst-case runtime of Mergesort is:

$$T(n) = 2T(n/2) + n$$

By the master theorem, $T(n) \in \Theta(nlogn)$.

Writing recurrence relations for D&C

Divide and Conquer algorithms naturally lead to recurrence relations, because the runtime for a given input size is typically a function of the runtime for some smaller input size.

Solving Recurrence Relations

"Unrolling" the recurrence

1. Repeatedly substitute in the definition of $T(n)$, searching for a pattern relating the recurrence and the depth, d
2. Come up with a formula for the recurrence expanded to an arbitrary depth
3. Solve for which d makes $n = 1$.
4. Plug that d back into the generalized recurrence, and simplify until you have a function $T(n)$ defined non-recursively.

Applying the master theorem

For a given recurrence:

$$T(n) = aT(n/b) + f(n)$$

Let our critical exponent $k = lg(a)/lg(b) = log_b(a)$.

1. If $f(n) \in O(n^{k-\epsilon})$ for some positive constant ϵ, then $T(n) \in \Theta(n^k)$.
2. If $f(n) \in \Theta(n^k)$, then $T(n) \in \Theta(n^k lg(n))$.
3. If $f(n) \in \Omega(n^{k+\epsilon})$ for some positive constant ϵ, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) \in \Theta(f(n))$.
    - The added constraint on f is called the regularity condition.

Maximum Sum Contiguous Subarray

Problem: given an array of numbers A, find the contiguous subarray with the maximum sum.

Naive solution

The naive solution, which involves checking every starting index and possible ending index, then summing, is $O(n^3)$.

Less naive solution

Initialize an array, `sums` such that `sums[i] = sum from 0...i, inclusive`. Now, calculating the sum of a contiguous subarray `(i, j)` is as simple as computing `sums[j] - sums[i]`.

D&C solution

1. **Divide:** Divide the array into two halves.
2. **Conquer:** Recurse on either half.
3. **Combine:** Step out, one index at a time, from the dividing line. Find the maximum contiguous subarray that crosses the dividing line.

Runtime of D&C solution

$$T(n) = 2T(n/2) + n$$

Just like Mergesort, $T(n) \in \Theta(nlogn)$.

Quicksort

A divide-and-conquer sorting algorithm, where most of the work is done during the "divide."

1. **Divide:** Select a "pivot" element, and call `partition` to sort every element around this pivot: they go below if less than the pivot, and above if greater than.
2. **Conquer:** Recursively call `quicksort()` on each partitioned side.
3. **Combine:** Nothing!

`partition` strategy

Inputs: an array `arr` and a pivot element `p`

```
def partition(arr, lb=0, ub=None):
    if ub is None:
        ub = len(arr)

    pivot = arr[ub - 1] # last element as pivot
    # if we want a different pivot, we just swap it there beforehand
    i = -1  # the rightmost element that's <= the pivot
    j = 0  # the rightmost element that's > the pivot
    while j < ub - 1:
        elem = arr[j]  # look at the jth element
        if elem <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
        j += 1

    arr[i+1], arr[ub-1] = arr[ub-1], arr[i+1]
    return i+1
```

Partition performs $n - 1$ comparisons when called on an array of n elements.

Runtime

In quicksort's worst case, we always select a min or max as a pivot. Our worst-case recurrence:

$$T(n) = 2T(n-1) + T(0) + n - 1$$

This recurrence is $\Theta(n^2)$, which we think is pretty bad. But it turns out this is unlikely in practice.

In the best case, quicksort always selects the median as a partition. The recurrence is:

$$T(n) = 2T(n/2) + n - 1$$

Just like mergesort, this is $\Theta(nlogn)$.

Considerations

The worst-case arises naturally if you give Quicksort (as described) an already-sorted list. The last element will always be a max, so the pivot will always be worst.

With proper precautions, the worst-case runtime is very unlikely. In the average case, quicksort performs very well. A few ways to avoid quadratic behavior:

- Select pivots randomly, or using some algorithm like Quickselect
- Approximate the median (sample 3 elements and pick middle?)

Properties

Quicksort is:

- In-place (if you don't consider the space complexity of a recursion stack)
- Unstable (although this depends on the partitioning algorithm)

Quickselect

Useful algorithm for finding a good pivot; quickselect finds the i^{th} order statistic in a list. (That is, the i^{th} smallest element in that list.)

Strategy: To find the i^{th} order statistic:

1. **Divide:** select an element p, and `partition(p)`
2. **Conquer:** if `i == return value of partition`, then done! Return p.
   - if `partition(p) < i`, recurse left (the i^{th} order statistic must be left of the split); else, recurse right
3. **Combine:** nothing

Runtime

If the pivot is always the median, then quickselect runs in $\Theta(n)$.

If it's always unbalanced, then quickselect runs in $\Theta(n^2)$.

Finding a good pivot for Quickselect

Median of medians guarantees a "pretty good pivot"—a value that's greater than 30% of the elements and less than 30% of the elements.

Strategy:

1. Break list into chunks of size 5
2. Find the median of each chunk using insertion sort
3. Return median of the found medians (using quickselect (!))

Using median of medians properly guarantees a $\Theta(n)$ runtime for quickselect.

Which guarantees a $\Theta(nlgn)$ runtime for quicksort.

In practice

In practice, it's actually better (probabilistically) to use a random pivot instead of using Quickselect.