1 Introduction

These notes introduce basic runtime analysis of algorithms. We would like to be able to tell if a given algorithm is time-efficient, and to be able to compare different algorithms.

2 Linear Search and Worst-Case Analysis

Let's say we have a list `L` of n integers (i.e., `len(L) == n`), and we'd like to find out if `e` is in the list or not. In Python, we would use:

```
e in L
```

to find out.

How might Python do this? If nothing is known about the list, Python would have to compare `e` to each element of `L`:

```
def linear_search(e, L):
  for e2 in L:
    if e == e2:
      return True
  return False
```

This algorithm is known as *linear search*.

How many iterations of the `for` loop will Python perform? It depends. It could be that `e == L[0]`, and `linear_search` will return during the first iteration. In the worst case, though, `e` is not in `L`, and Python would have to check every element of `L` to make sure of that. So, n = `len(L)` iterations of the loop will be performed before `linear_search` returns.

When we do analysis of algorithms, we usually (though not always) concentrate on the analysis of the **worst-case** runtime. That's mainly because we would like to be able to make a guarantee that our program will finish within a certain timeframe[1].

To get the worst-case number of iterations, we needed to come up with an example of input values (`L` and `e`, in our example) when the worst case happens. Note that in this example, the worst case is for lists of length n only – obviously longer lists would take longer to process. If the input includes lists, dictionaries, or other structures that vary in size, we usually fix the size of the input (in this case, the length of `L`), and figure out what the worst-case input of any given size is. The number of iterations of **an appropriate loop** is proportional to the runtime (in seconds). Instead of counting the number of iterations, we might also count, for example, the number of comparisons performed. (In the case of linear search, the number of comparisons performed explicitly is the same as the number of iterations).

3 Binary Search

Is there a faster way to search the list `L`? There isn't if `L` is an arbitrary list – the only way to make sure that `e not in L` is to go through every element of `L` and check. However, if we know that `L` is sorted, we can get away with not checking every element.

[1] It's clear that thinking about the best case doesn't make a lot of sense, but thinking about the average case – how long the program takes *on average* (the average would be taken over the runtimes for different inputs) does make sense sometimes, and is the only option that makes sense for some algorithms.

Here's why: if we know that `e > L[len(L)//2]` and that `L` is sorted, there's no point in looking in the left half of `L`: `e` is definitely not there. Similarly, if `e < L[len(L)//2]`[2] we can avoid looking in the right half of `L`.

So by checking whether `e > L[len(L)/2]`, we can effectively halve the size of the list through which we need to look. We can then use the same trick to half the size of the remaining list. This algorithm is called *binary search*, and it's the same algorithm that people use when playing Twenty Questions.[3] In out case, we might ask: "If `e` is in `L`, is it in the first half or the second half?" If we hear "Second," we ask: "'If `e` is in `L`, is `e` in the third quarter of `L` or the fourth quarter of `L`," and so on, until we finally zero in on the only possible location of `e` in `L`, at which point we can finally check whether it's there or not.

Here's an implementation of binary search which returns the index of the element `e` if it is found:

```
def binary_search(L, e):
    low = 0
    high = len(L)-1
    while high-low >= 2:

        #2 comparisons
        mid = (low+high)//2   #e.g. 7//2 == 3
        if L[mid] > e:
            high = mid-1
        elif L[mid] < e:
            low = mid+1
        else:
            return mid


    if L[low] == e:
        return low
    elif L[high] == e:
        return high
    else:
        return None
```

In the worst case, we don't return until we are done with the `while` loop. That happens if `L` doesn't contain an element that's equal to `e`. How many comparisons will we have in that case? We do `3` comparisons per iteration of the `while` loop, plus, in the worst case, another `2` comparisons before returning after the `while` loop.

For a list `L` of length n, how many comparisons are performed in total, in the worst case ? It's $2+3\times$(the number of iterations of the while loop).

How many iterations of the while loop are there, in the worst case? We think about this by figuring out the length of the sublist still under consideration at every iteration. Approximately, every iteration, the sublist under consideration becomes smaller by a factor of 2: $n, n/2, n/4, n/8, ..., 1$. To get from 1 to n

[2] We use // to always get an integer index. Recall that // in Python 3 is integer division. For example, 17//2 == 8.

[3] in Twenty Questions, person A thinks of something or someone, and person B needs to ask at most 20 yes-or-no questions before taking a guess at what or who person A is thinking of. For example, the game might go: "Is it a person?" "Yes." "Is it a celebrity?" "Yes." "Are they currently alive?" "Yes." "Is it a woman?" "No." "Is he an artist?" "Yes." "Is he a painter?" "No." "Is he a musician?" "Yes." "Is he from North America?" "Yes." "Is he from the U.S.?" "No." "Is he Canadian?" "Yes." "Is he older than 30?" "No." "Is it Justin Bieber?" "Yes."

by multiplying by 2, it takes $\log_2 n$ iterations (by the definition of log!). So, in the worst case (when `e` is not in `L`), for a list `L` of length n, binary search performs approximately $2 + 3\log_2 n$ comparisons.

4 Big-Oh

At the end of the day, we don't care about the *number of iterations* of the *number of comparisons*. What we care about is how long – in seconds – a program will take to run. If we choose the things we count – iterations of a certain loop, or operations–well, their number will be proportional to the runtime of the program in seconds. Because our answer (when the unit is iterations, or number of operations) is merely proportional to the runtime, we can drop multiplicative constants. Because we care only about what happens when the input is large, we only retain the fastest-growing term in the expression.

Dropping the multiplicative constant 3 and the slower-growing term 2 (2 doesn't grow at all!), we say that binary search runs in $\mathcal{O}(log(n))$ time (note: $\log(n)$ is proportional to $\log_2(n)$, so we drop the 2 as well.

The Big-Oh notation ($g(n)$ is $\mathcal{O}(f(n))$) is used to express an upper bound on what the function $g(n)$ is proportional to, for large n.

Technically, the function $g(n)$ is $\mathcal{O}(f(n))$ if $\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$. In English, this means that $g(n)$ grows at most as fast as $f(n)$ for large n. We don't need to know the technical definition. The important parts are:

- $g(n)$ is $\mathcal{O}(ag(n))$ for any positive constant a.
- $g(n) + h(n)$ is $\mathcal{O}(g(n))$ if $h(n)$ grows at most as fast as $g(n)$ for large n (i.e., $h(n)$ is $\mathcal{O}(g(n))$.

This lets us present the runtimes of algorithms in a simplified form:

- $2n + 5$ is $\mathcal{O}(n)$.
- $\frac{1}{2}n^3 + 5n$ is $\mathcal{O}(n^3)$.
- $3n\log_{10} n + 10000n$ is $\mathcal{O}(n\log n)$.

Note that it is also true to say that $500n$ is $\mathcal{O}(n^2)$, since n^2 grows faster than $500n$. We prefer to give runtime upper bounds that are as *tight* as possible, so if the algorithm runs in time proportional to $500n$, we'll say that it runs in $\mathcal{O}(n)$ time (even though technically, it also runs in $\mathcal{O}(n^5)$ time),

5 Algorithm Runtime Analysis: The Steps

- Pick a unit in which you're measuring the runtime. The number of those units should be proportional to the actual runtime in seconds.
- Pick a property of the input (e.g., the value, or the length). You'll be computing the runtime as a function of that property.
- Compute the runtime, in the worst case, in your chosen units
- Simplify the expression

5.1 Units using which we analyze the algorithm

What we really want to measure is the number of seconds in which the algorithm runs. However, we cannot directly compute that number. Instead, we count the *the number of basic operations*, or the number of *iterations* of the innermost loops, or the *number of comparisons performed*, or the *number of arithmetic*

operations performed. The important thing is to make sure that the thing we're counting is actually proportional to the runtime – we need to use our common sense here, and also keep in mind that in Python, some seemingly basic operations can take a lot of time.

For example,

```
e in L
```

is not a basic operation. The only way to return `True` or `False` there is to check over every element of `L` and compare it to `e`, in the worst case (in the best case, `e == L[0]`, so we can return `True` pretty fast, since we'll discover that `e in L` pretty fast.) So `e in L` actually runs in $\mathcal{O}(n)$ for `n = len(L)`.

For sorting algorithms, it makes sense to pick comparisons as the unit (by comparisons, we mean operations like `==`, `<`, `>=`, etc, since the number of those is typically proportional to the runtime.

For simple nested loops, we can generally get away with claiming that the number of times that the innermost loop repeats is proportional to the runtime. For example, consider the following.

```
s = 0
for i in range(n):
    for j in range(n):
        s += i*j
```

The innermost loop repeats n times every time the outer loop runs, so the innermost loop repeats n^2 times in total, so $\mathcal{O}(n^2)$ times. The runtime is also proportional to $\mathcal{O}(n^2)$.

5.1.1 Summary

We usually would like to count how many times either the innermost loop runs, or how many basic operations are performed. Those quantities should be proportion to the runtime in seconds.

5.2 Property of the input based on which we calculate the runtime

The kind of question that we want to ask is: "In the worst case, for a list `L` of length n, what is the runtime of sorting `L` using bubble sort?" We might also ask, "In the worst case, for a list `L`, what is the runtime for sort it using bubble sort?"

In the first case, the answer is $\mathcal{O}(n^2)$, when n is `len(L)`. In the second case, we might say that the runtime is 3 comparisons, for `L = [1,2,3]`, 6 comparisons for `L = [3,2,1]`, etc. (Note: those numbers were made up.)

It is a decision we make that we analyze the runtime as a function of the *length* of `L`. We could, in principle, also decide to analyze the runtime based on the precise value of `L` (e.g., `[1,2,3]`, or `[3,2,1]`).

For lists, we almost always analyze the runtime as a function of the length of the list. In that case, it makes sense to talk about the *worst-case* runtime, since the runtime can be different for two different lists which are both of length `n`.

For integers, we sometimes analyze the runtime as a function of the *value* of n, and sometimes as a function of the *length* of n. The length of the integer is just the number of (binary) digits that we need to write it down. To write down n, we need about $\log_2 n + 1$ binary digits. Since $\log_1 0(n) \propto \log_2(n) \propto \log(n)$, and since we generally ignore additive constants, we sometimes will analyze the runtime of `func(n)` as a function of $\log(n)$, a quantity proportional to the length of n.

5.2.1 Summary

For lists, we usually want to get the logest possible runtime for a list of length n, as a function of n: $\mathcal{O}(n)$, $\mathcal{O}(n \log(n))$, etc.

5.3 Compute the runtime, in the worst case, in your chosen units

There is no general technique for computing the runtime. For lists, we need to come up with an example of a list of length n such that the algorithm will run the longest.

5.4 Simplifying the expression

When thinking about the worst-case runtime of the algorithm on input of size/value n, we care about a *tight upper bound* on what the runtime is approximately proportional to for large n. That means that if we have an expression like $n^2 - n + 15$, we drop any terms that grow slower than the fastest-growing term, and any multiplicative constants. If the algorithm runs in time proportional to $n^2 - n + 15$, we say it runs in $\mathcal{O}(n^2)$ time.