Improving Algorithmic Efficiency

15-112

Big Ideas

Efficiency in Algorithms

Now that we know how to calculate the efficiency of a program, we need to consider efficiency during **algorithm design**.

Small changes in a program can lead to big changes in runtime!

Practical Efficiency Improvement

When trying to improve the efficiency of an algorithm, you can consider several factors.

- Is it doing an action multiple times that could instead be done only once?
- Would the data be better represented in a different data structure?
- Is the function breaking out of loops and returning as soon as it can?

These might not always improve the function family of an algorithm, but they can still make substantial changes.

Improvement Example - multiple actions

```python
def maxIndexes(lst):
    result = []
    for i in range(len(lst)):
        if lst[i] == max(lst):
            result.append(i)
    return result
```

```python
def maxIndexes(lst):
    result = []
    maxVal = max(lst)
    for i in range(len(lst)):
        if lst[i] == maxVal:
            result.append(i)
    return result
```

Improvement Example - data structure

```
def mostCommonItem(lst):
    bestItem = None
    bestCount = 0
    for item in lst:
        if lst.count(item) > bestCount:
            bestItem = item
            bestCount = lst.count(item)
    return bestItem
```

```
def mostCommonItem(lst):
    countD = { }
    for item in lst:
        countD[item] = countD.get(item, 0) + 1
    bestItem = None
    bestCount = 0
    for item in countD:
        if countD[item] > bestCount:
            bestItem = item
            bestCount = count
    return bestItem
```

Improvement Example - returning early

```python
def isPrime(n):
    prime = True
    if n < 2:
        prime = False
    for factor in range(2, n):
        if n % factor == 0:
            prime = False
    return prime
```

```python
def isPrime(n):
    if n < 2:
        return False
    for factor in range(2, n):
        if n % factor == 0:
            return False
    return True
```

Algorithmic Efficiency: Searching

Activity: Find a Word in a Book

Linear Search

In linear search, we methodically look at **every element in the list**.

This approach is good when the element could be **anywhere** in the list.

Let’s code it!

Binary Search

In binary search, we search within a **subset of the list** where we know the item might be. This subset starts as the whole list. We then compare the middle element to our item.

- If it is our item, we're done!
- If it's smaller than our item, change the **left bound** of the subset to the middle index + 1
- If it's bigger than our item, change the **right bound** of the subset to the middle index - 1

Then keep going until we find the item, or until the subset is empty.

This approach is good when we **know that the list is sorted**.

Let's code it!

Can we do better?

In linear search, we look at every item- that's **O(N)**.

In binary search, we keep halving the size of the list until it's empty- that's **O(logN)**.

Is there a way for us to search for an item in better than O(logN) time?

Sets & Dictionaries: Hashing

How do we make super-efficient datatypes?

We know that sets and dictionaries let us look up (search) an element in **constant time**.

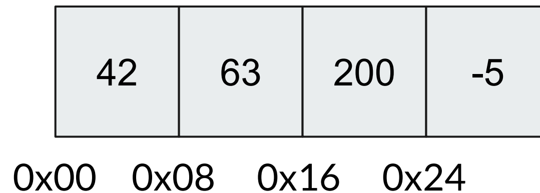
How is that possible? We just showed that searching a list takes O(logN) time, and that’s if it’s sorted!

List Representation

A list maps indexes from 0 to N to values of any type.

In memory, these values are then stored side-by-side in equal-sized ‘bins’. The list also keeps track of the starting position.

```
lst = [ 42, 63, 200, -5 ]
```



List Representation - Lookup

This representation lets us determine the location of an index with a simple formula:

```
startLocation + index * binSize
```

This means we can look up the value at a specific **index** in constant time!

Finding the index of a specific **value** still takes linear time- we have to check all possible indexes.

```
lst = [ 42, 63, 200, -5 ]
lst[2] # 0x00 + 2*8 = 0x16
```

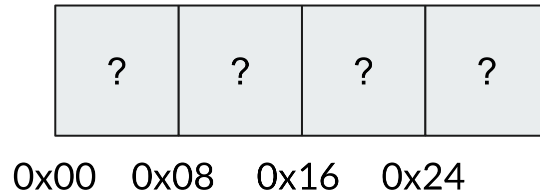
42	63	200	-5
0x00	0x08	0x16	0x24

Set representation

Sets don't have indexes that we can see. However, in the implementation of sets, values are stored in a secret list of some size that *does* have indexes.

How does a set determine which index a value should go to? **Use the value itself!**

```
s = { 42, 63, 200, -5 }
```



Hash Functions

A **hash function** is a function that maps a value to an integer. This function must have two properties:

1. f(value) should return the same number **every time** it's called on the same value
2. f(value) should generally return different numbers for different values (though they can occasionally be the same)

Python has its own built-in hash function: `hash(value)`

Set Representation

To find the needed index of a given value, a set computes `hash(value)`.

However, that number might be out of bounds of the list. Therefore, the final index is:

```
hash(value) % len(list)
```

What if multiple values have the same index? Put them all at that index, in an inner list.

```
s = { 42, 63, 200, -5 }
# indexes: 2, 3, 0, 3
```

200		42	63, -5
0x00	0x08	0x16	0x24

Set Representation - Lookup

When we want to see if a value is in a set, we don't need to look at every possible index.

Instead, **re-compute the value's index** using the hash function.

If the value isn't at that location, it isn't in the list!

Because we only have to check one index, and because we can make the underlying list as large as it needs to be, this is **constant time**.

```
s = { 42, 63, 200, -5 }
200 in s # hash(200) % 4 = 0, check l[0]
73 in s # hash(73) % 4 = 1, check l[1]
```

200		42	63, -5

0x00 0x08 0x16 0x24

Sets and Mutability

Sets have one major restriction: **they can only hold immutable values**.

Why? Consider the situation on the right. What could go wrong?

To avoid this situation, calling hash on a mutable value or adding a mutable value to a set will raise an error.

```
s = set()
lst = [1,2,3]

s.add(lst)

lst.append(4)

print(lst in s)
```

Algorithmic Efficiency: Sorting

Many Ways to Sort

As we've discussed before, there are often multiple different ways to solve the same problem. This is especially true for the problem of sorting an unordered list. In fact, hundreds of different sorting algorithms exist!

We'll focus on three: **bubble sort, selection sort**, and **merge sort**. These algorithms provide a good case study of why algorithm design matters in efficiency. You can find code for each of these on the website.

Bubble Sort

Idea: while the list isn't sorted, compare each sequential pair of elements, and swap them if they're out of order. If you make an entire pass through the list without swapping, it's sorted!

Example: http://math.hws.edu/eck/js/sorting/xSortLab.html

Bubble Sort Function Family

Instead of looking directly at the code, let’s consider the algorithm at a high level. We’ll mainly consider the algorithmic steps of **swaps** and **comparisons**.

In each iteration, the algorithm makes K-1 comparisons + up to K-1 swaps (where K is the number of unsorted elements).

How many iterations happen? In the worst case, we’ll have to iterate once for each element- N times.

That’s $2*(N-1 + N-2 + N-3 + \ldots + 3 + 2 + 1) \rightarrow 2*(N-1)*(N-1)/2$.

Function Family: **O(N**2).** Does that match the code?

Selection Sort

Idea: look through all the elements that haven’t been sorted yet, keeping track of the index of the smallest one; then swap that element with the first unsorted index.

Example: http://math.hws.edu/eck/js/sorting/xSortLab.html

Selection Sort Function Family

In each iteration, we do K-1 comparisons and 1 swap (where K is the number of unsorted elements).

How many iterations happen? Exactly N- one for each moved element.

Again, N + N-1 + N-2 + ... + 2 + 1 -> N*N/2

Function Family: **O(N**2)**. Does that match the code?

Merge Sort

Idea: Instead of swapping elements, we start by noting that a list of length 0 or 1 is sorted. We go through the list and **merge** pairs of sorted sublists into sorted lists of length 2 by moving them in sorted order into a temporary list, then back to the original list. We then repeat for length 4, then 8, etc., until the whole list is sorted.

Example: http://math.hws.edu/eck/js/sorting/xSortLab.html

Merge Sort Function Family

First: what is the runtime of the **merge** step? To merge two lists each of length N, we need to do N-1 comparisons, then move N elements from the temp list to the original list. That means merging all the pairs of sublists in a list of length N takes N+1 + N -> **O(N)** time.

Second: how many iterations occur? Each time we run the merge step, we **double** the size of the sorted sublists. This means we have to run merge the number of times it takes to divide N by 2- in other words, **O(logN)**.

Function Family: **O(NlogN).** [Does that match the code?]

Sorting Efficiency

Fun fact: **O(NlogN) is the best generic sorting efficiency possible**, at least so far.

Can we ever do better? Yes- sometimes!

- If we parallelize the sorting work, we can run in O(N) time (though still with O(NlogN) work).
- If the elements of the list can be mapped to integers, we can also sort in O(N) time with a method similar to hashing.
- If we get lucky and get a good input, some algorithms run in O(N), including Bubble Sort.