CSCI 134 Fall 2021:

Searching & Efficiency

Nov 22, 2021

Shikha Singh, 9AM
Jeannie Albrecht, 10AM
Announcements & Logistics

- **Lab 8** feedback coming soon!
- Lab 9 part 1 feedback returned: let us know if you have any questions!
- **No homework** this week
- **Lab 9 Boggle**
  - Today/tomorrow lab: attendance optional (but encouraged)
  - **Parts 3 & 4 (BoggleWords & Game)** are due Dec 1/2 next week after Thanksgiving break
- No TA hours after Monday 11/22 evening

**Do You Have Any Questions?**
Last Time: Iterators

- Learned about **iterables**, **iterators**, and **generators**

- An object is considered **iterable** if it supports the `iter()` function (special method `__iter__` is defined): e.g., lists, strings, tuples

- When an **iterable** is passed to the `iter()` function, it creates and returns an **iterator**

- An **iterator** object can generate values **on demand**
  - **Supports the** `next()` **function (and** `__next__` **method)** which simply provides the "next" value in the sequence

- There are two ways to create iterators in Python:
  - **Method 1:** Define `__iter__` and `__next__` special methods
  - **Method 2:** Via **generators**
Recap: Generators

- Any function with a `yield` statement is a **generator**
- When a generator function is invoked, it returns a **generator object** (which is a type of **iterator**)
- When `next()` is called on that generator object, execution continues until a `yield val` statement is reached
  - Then execution is **paused**, and `val` is returned (or yielded)
  - Subsequent calls to `next()` continue execution
- We implemented `__iter__` as a generator in `LinkedList` to yield next value

```python
def __iter__(self):
    # set current to head
    current = self
    # while current is not None
    while current:
        yield current.value
        current = current.rest
```
Today and Next Week

• Briefly introduce how we measure efficiency in Computer Science
• Analyze the efficiency of some of our algorithms and data structures
• Next Monday:
  • Evaluate sorting algorithms and their efficiency
• Last 5 classes: Introduction to Java
  • Computational thinking and logic stays the same across programming languages
  • We will focus on how the two languages are different in their syntax and structure
Measuring Efficiency

• How do we measure the efficiency of our program?
  • We want programs that run "fast"
  • But what do we mean by that?
• One idea: use a stopwatch to see how long it takes
  • Is this a good method?
  • What is the stopwatch really measuring?
  • How long does this piece of code takes on this machine on this particular input.
• Machine dependent
  • We want to evaluate our program’s efficiency, not the machine's speed
• Cannot make any general conclusions
  • Might not tell us how fast the program will run on different inputs
Efficiency Metric: Goals

We want a method to evaluate efficiency that:

- **Is machine and language independent**
  - Analyze the *algorithm* (problem-solving approach) instead

- **Provides guarantees that hold for different types of inputs**
  - Some inputs may be "easy" while others are not

- **Captures the dependence on input size**
  - How the performance "scales" when the input gets bigger

- **Captures the right level of specificity**
  - We don't want to be too specific (cumbersome)
  - Measure things that matter, ignore what doesn't
Platform/Language Independent

**Machine and language independence**

- We want to evaluate how good the algorithm is, rather than how good the machine or implementation language is.
- Basic idea: Count the number of steps taken by the algorithm.
- Sometimes referred to as the "running time" (abusing term).
Worst-Case Analysis

• We can't just analyze our algorithm on a few inputs and declare victory

  • **Best case.** Minimum number of steps taken over all possible inputs of a given size

  • **Average case.** Average number of steps taken over all possible inputs of a given size

• **Worst case.** Maximum number of steps taken all possible inputs of a given size.

• Benefit of worst case analysis:

  • Regardless of input, can conclude that algorithm always does at least as well as the pessimistic analysis.
Dependence on Input Size

- We generally don't care about performance on "small inputs"
- Instead we care about "the rate at which the time taken grows" with respect to the input size
- For example, consider the area of a square or circle: while the formula for each is different, they both grow at the same rate wrt radius
  - Doubling radius increases area by 4x, tripling increases by 9x

Doubling $r$ increases area $4 \times$. Tripling $r$ increases area $9 \times$. 
Dependence on Input Size: Big Oh

• Big-O notation in Computer Science is a way of quantifying (in fact, upper bounding) how the function grows wrt input size

• For example:
  • A square of side length $r$ has area $O(r^2)$.
  • A circle of radius $r$ has area $O(r^2)$.

Doubling $r$ increases area $4 \times$.
Tripling $r$ increases area $9 \times$. 
Dependence on Input Size: Big Oh

- Big-O notation captures the rate at which the number of steps taken by the algorithm grows with respect to the size of input \( n \), "as \( n \) gets large"

- Not precise by design, it ignores information about:
  - Constants (that do not depend on input size \( n \)), e.g. \( 100n = O(n) \)
  - Lower-order terms: terms that contribute to the growth but are not dominant: \( O(n^2 + n + 10) = O(n^2) \)

- Powerful tool for predicting performance behavior: focuses on what matters, ignores the rest

- Separates fundamental improvements from smaller optimizations

- We won't study this notion formally: covered in CS136!
Understanding Big-O

- Notation: \( n \) often denotes the number of elements (size)

- **Constant time** or \( O(1) \): when an operation does not depend on the number of elements, e.g.
  - Atomic operations: addition/subtraction/multiplication or two values, or defining a variable etc are all constant time

- **Linear time** or \( O(n) \): when an operation requires time proportional to the number of elements, e.g.:
  ```python
  for item in seq:
      <do something>
  ```

- **Quadratic time** or \( O(n^2) \): nested loops are often quadratic, e.g.,
  ```python
  for i in range(n):
      for j in range(n):
          <do something>
  ```
Big-O: Common Functions

- Notation: \( n \) often denotes the number of elements (size)
- You will study these in more detail in CS136
- Our goal: understand efficiency of some algorithms at a high level
Arrays vs Linked Lists:
Efficiency Trade Offs
**Array vs Linked Lists**

- **Linked Lists**: pointed-based data structure, items need not be contiguous in memory

  ![Linked List Diagram](image)

- **Arrays**: index-based data structure, items stored contiguously in memory

  ![Array Diagram](image)
Array vs Linked Lists

- **Linked Lists**: Can grow and shrink on the fly: do not need to know size at the time of creation (no wasted space!)

- **Arrays**: Need to know size at the time of creation, can waste space by leaving room for insertions (can be made dynamic, but let’s ignore for now)
What is Python's list?

- It's complicated: Python's implementation is a hybrid
  - For smaller sizes, it more like a (dynamic) array
- For today's lecture, we will assume its an array
Array vs Linked Lists

- Inserts at the beginning: which one is better?
Array vs Linked Lists

- Linked list steps:
  - Point head to new element
  - Point rest of new element to old list
  - Steps don't depend on size of list
  - Therefore, run-time is constant, that is, $O(1)$ time
Array vs Linked Lists

• Python’s list is an optimized version of a dynamic array (grows and shrinks as needed)

• It’s more complicated than this but let’s ignore that for now

• To insert at index 0, we need to shift every element over by one spot
  • This takes time proportional to the size: linear time or $O(n)$

• When can arrays be better?
  • When indexing elements: they give direct access $O(1)$
  • Linked list: we need to traverse the list to get to the element $O(n)$
So Which is Better?

• It depends!

• **Time-space tradeoff**: try to find a balance between *time efficiency* and *space efficiency*

• Think about what list operations are required the most for your program

• Choose accordingly
Searching in an Array
Searching in an Array

• For now assume that Python's list is implemented as an array
• Let us discuss how quickly we can search for an item in it

```python
def linearSearch(e, L):
    for elem in L:
        if elem == e:
            return True
    return False
```

Might not always run, but assume it does: overestimate

Might return early if e is first item in list but interested in the worst case; happens if e is not in the list or last item
Searching in an Array

• In the worst case, we have to walk through the entire sequence

• Takes linear time, or $O(n)$

```python
def linearSearch(e, L):
    for elem in L:
        if elem == e:
            return True
    return False
```

Might not always run, but assume it does: overestimate

Might return early if $e$ is first item in list but interested in the worst case; happens if $e$ is not in the list or last item

```
8  5  3  11  ...
```

```
0  1  2  3
```
Searching in an Array

• Can we do better?
  • No, if the elements are in arbitrary order
• What if the sequence is sorted?
  • Can we utilize this somehow and search more efficiently?

How do we search for an item (say 10) in a sorted array?
Example: Dictionary

• How do we look up a word in a (physical) dictionary?

• Words are listed in alphabetical order
Example: Dictionary

• How do we look up a word in a (physical) dictionary?

• Words are listed in alphabetical order

Let’s assume we don’t have these tabs to help us out
Searching for Word in Dictionary

• Look at the (approximately) middle page for our query word
• If we find our query, great!
• Otherwise:
  • If our query is later in alphabetical order than the words on the page, look for the query between the middle page and the last page
  • If our query is earlier in alphabetical order, look for the query between the middle page and the first page
How Good is This Method?

• **Goal:** Analyze # pages we need to look at until we find the word

• We want the worst case: possible to get lucky and find the word right on the middle page, but we don't want to consider luck

• Each time we look at the “middle” of the remaining pages, the number of pages we need to look at is divided by 2

• A 1024-page dictionary requires at most 11 lookups:
  1024 pages, < 512, <256, <128, <64, <32, <16, <8, <4, <2, <1 page.

• Only needed to look at 11 pages out of 1024!

• What if we have an $n$ page dictionary, what function of $n$ characterizes the (worst-case) number of lookups?
Logarithms: CS's favorite function

- Logarithms are the inverse function to exponentiation.
- $\log_2 n$ describes the exponent to which 2 must be raised to produce $n$.
- That is, $2^{\log_2 n} = n$.
- Alternatively:
  - $\log_2 n$ (essentially) describes the number of times $n$ must be divided by 2 to reduce it to below 1.
- For us, here's the important takeaway:
  - How many times can we divide $n$ by 2 until we get down to 1?
  - $\approx \log_2 n$
Binary Search

• The **recursive search algorithm** we described to search in a sorted array is called **binary search**

• It is much, much more efficient than a **linear search**: $O(\log n)$ time
  
  • **Note:** $\log n$ grows much more slowly compared to $n$ as $n$ gets large

• Lets implement this technique

```python
def binarySearch(aList, item):
    """Assume aList is sorted.
    If item is in aList, return True;
    else return False.""
    pass
```
Binary Search

- Base cases? When are we done?
  - If list is too small (or empty)
  - If item is the middle element

```python
def binarySearch(aList, item):
    """Assume aList is sorted.
    If item is in aList, return True;
    else return False.""
    n = len(aList)
    mid = n // 2
    # base case 1
    if n == 0:
        return False
    # base case 2
    elif item == aList[mid]:
        return True
```
Binary Search

- Recursive case:
  - Recurse on left side if item is smaller than middle
  - Recurse on right side if item is larger than middle

If item < L[mid], then need to search in L[:mid]
Binary Search

- Recursive case:
  - Recurse on left side if item is smaller than middle
  - Recurse on right side if item is larger than middle

If item > L[mid], then need to search in L[mid+1:]
def binarySearch(aList, item):
    """Assume aList is sorted. If item is in aList, return True; else return False."""
    n = len(aList)
    mid = n // 2
    # base case 1
    if n == 0:
        return False

    # base case 2
    elif item == aList[mid]:
        return True

    # recurse on left
    elif item < aList[mid]:
        return binarySearch(aList[:mid], item)

    # recurse on right
    else:
        return binarySearch(aList[mid + 1:], item)