Introduction to Algorithm Analysis
Part 1

David Fernández-Baca

Computer Science 311
Iowa State University

January 20, 2022

The Search Problem

Given: An array of integers $A[0 \mathinner{.\,.} n-1]$ and an integer v.

Return: `true` if there is an index i such that $A[i] == v$; `false` if no such index exists.

Variation

Instead of `true`/`false`, return index i such that $A[i] == v$, or -1 if no such index exists.

```
SequentialSearch(A, v)
    Input: An array A and a value v.
    Output: true if there is an index i such that A[i] == v; false if
            no such index exists.
    n = A.length
    for i = 0 to n − 1 do
        if A[i] == v then
            return true
    return false
```

Evaluating Algorithms

- Correctness
- Speed/running time
- Amount of memory
- Elegance/simplicity
- Ease of implementation
- Reusability

CS 311 focuses on the first three.

Evaluating the Running Time of an Algorithm

Naïve Approach

Implement the algorithm and time it on different inputs.

- Alternatively: count CPU cycles.

Evaluating the Running Time of an Algorithm

Drawbacks of Naïve Approach

- Too dependent on implementation details and runtime environment

  CPU speed,
  memory speed,
  cache locality,
  garbage collection, etc.

- Implementation could be nontrivial.

  Better to study efficiency before committing time and money to coding.

- Says little about how an algorithm scales as we increase the input size or when we get a faster machine.

Running Time

Definition

The running time of an algorithm is a function that describes the number of basic execution steps in terms of the input size.

Idea

Running time abstracts the components of an algorithm's performance that depend on the algorithm itself away from those components that are machine- and implementation-dependent.

What is a "basic execution step"?

- For the analysis to correspond usefully to the actual execution time, the time required to perform a basic step must be guaranteed to be bounded above by a constant.
- Typically, assume the following operations take constant time:
  - Assignments
  - Arithmetic: addition, subtraction, multiplication, division
  - Comparisons
- Be careful. E.g., if the numbers involved in an addition are large, we cannot assume the operation takes constant time.

Cost Models

Uniform Cost Model

Each operation has a constant cost, regardless of the size of the numbers involved.

- Simple and widely used.

  We use it by default in CS 311.
- May be unrealistic if numbers involved are large.

Cost Models

Logarithmic Cost Model

Cost of each operation is proportional to the number of bits involved.

- More precise, but more cumbersome than uniform cost model.
- Employed when necessary, e.g., in the analysis of arbitrary-precision algorithms in cryptography.

In CS 311, unless otherwise specified, we use the uniform cost model.

Types of Algorithm Analysis

- Best case. Running time on "easiest" input of size n.
- Worst case. Running time guarantee for any input of size n.
- Probabilistic. Expected running time of a randomized algorithm.
- Amortized. Worst-case running time for any sequence of operations.
- Average case. Running time on "average" input of size n.
  - Requires knowledge about the distribution of inputs.

We will focus on worst-case analysis, as it generally captures efficiency in practice.

Worst-case analysis of `SequentialSearch`

- Worst-case: All elements of A are scanned and v is not found.
- Assume each basic step takes at most c time.
- c depends on programming language, compiler, machine, OS, etc.
- The worst-case time is

$$T(n) \leq \underbrace{cn}_{n \text{ comparisons}} + \underbrace{2c}_{\text{initializing } n \text{ and } \textbf{return}} .$$

- Regardless of the value of c, we can say that the running time is linear in n.
- Formally, we say that $T(n)$ is $O(n)$.
- Even more precisely, $T(n)$ is $\Theta(n)$.

Exercise

```
CheckDuplicates(A, B)
    Input: Arrays A and B, each containing n integers.
    Output: true if there is an integer v that appears in both A and
            B; false otherwise.
    for i = 0 to n − 1 do
        for j = 0 to n − 1 do
            if A[i] == B[j] then
                return true
    return false
```

Question

What are the best- and worst-case running times of `CheckDuplicates` as a function of the input size, n?

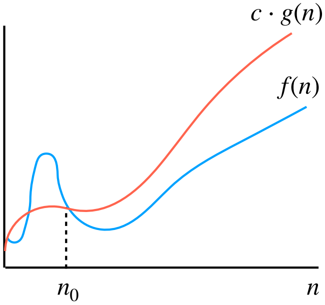
O-Notation

O-notation

Definition

$f(n)$ is $O(g(n))$ if and only if there exist positive constants c and n_0 such that

$$f(n) \leq c \cdot g(n), \quad \text{for all } n \geq n_0.$$



$f(n) = O(g(n))$

O-notation

- $f(n)$ is $O(g(n))$ if we can multiply $g(n)$ by a (possibly large) constant c so that, asymptotically (as $n \to \infty$), $f(n)$ is completely underneath $c \cdot g(n)$.
- Equivalently, $f(n) = O(g(n))$ if and only if there exists a constant $c \geq 0$ such that
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c.$$

Example 1

Proposition

$f(n) = 3n + 3$ *is* $O(n)$

Proof.

$$3n + 3 \quad \leq \quad 3n + n \quad \leq \quad 4n, \qquad \text{for } n \geq 3.$$

Hence, choose $c = 4$ and $n_0 = 3$. □

Example 2

Proposition

$f(n) = 5n + 45$ is $O(n)$.

Proof.

$$5n + 45 \quad \leq \quad 5n + n \quad \leq \quad 6n, \qquad \text{for } n \geq 45.$$

Hence, choose $c = 6$ and $n_0 = 45$. □

Example 3 (Generalization of Examples 1 and 2)

Proposition

Let $f(n) = an + b$, where $a > 0$. Then, $f(n)$ is $O(n)$.

Proof.

$$an + b \quad \leq \quad an + n \quad \leq \quad (a+1)n, \qquad \text{for } n \geq |b|.$$

Hence, choose $c = a + 1$ and $n_0 = |b|$. $\square$

Constant Factors and Big-O

- When using O-notation, keep things as simple as possible.
- In particular, ignore constant (multiplicative) factors!

- Let $f(n) = an + b$, where $a > 0$.
- We just proved that $f(n) = O(n)$.
- It is also true that $f(n) = O(2n)$.
- However, the constant 2 does not add any essential information.

O-notation is for upper bounds

Example

Suppose $f(n) = 3n^3 + 4n^2 + 10n + 12$. Then,

1. $f(n)$ is $O(n^3)$: choose $c = 29$, $n_0 = 1$
2. $f(n)$ is $O(n^4)$. (What would c and n_0 be?)
3. $f(n)$ is not $O(n)$.
4. $f(n)$ is not $O(n^2)$.

(1) and (2) are correct, since they're both upper bounds, but (1) is more precise (tighter).

Always aim to give the tightest O-bound possible.

Insertion Sort

Sorting

Input: An n-element array $A[0 \, . \, . \, n-1]$.

Goal: Rearrange elements of A so that

$$A[0] \leq A[1] \leq A[2] \leq \cdots \leq A[n-1].$$

Insertion Sort

Insertion Sort(A)

for $i = 1$ to $n - 1$:
insert $A[i]$ in its proper place amongst $A[0\,.\,.\,i]$.

Example

$$A = \langle 4, 3, 2, 1\rangle \to \langle 3, 4, 2, 1\rangle \to \langle 2, 3, 4, 1\rangle \to \langle 1, 2, 3, 4\rangle$$

```
InsertionSort(A)
    n = A.length
    for (i = 1; i < n; i++) do
        temp = A[i]
        j = i - 1
        while j > -1 and A[j] > temp do
            A[j + 1] = A[j]
            --j
        A[j + 1] = temp
```

Correctness of `InsertionSort`: Loop Invariants

Definition

A loop invariant is a statement that is initially true and remains true after each execution of a loop.

Example (An invariant for `InsertionSort`)

Insertion sort maintains the following invariant:

> *At the start of iteration i of the* **for** *loop, $A[0\,..\,i-1]$ consists of the elements originally in $A[0\,..\,i-1]$, but in sorted order.*

Correctness of `InsertionSort`: Loop Invariants

Loop invariants provide a way to prove the correctness of algorithms.

Example (Correctness of Insertion Sort)

- (Initialization) The invariant is true at the outset.
- (Maintenance) The loop maintains the invariant through shifting and insertion.
- (Termination) At termination, $i = n$, so the invariant implies that subarray $A[0 \,..\, n-1]$ — i.e., the whole array — consists of the elements originally in $A[0 \,..\, n-1]$, but in sorted order.

The last statement proves the correctness of insertion sort.

Analysis of `InsertionSort`

- The **for** loop iterates $n-1$ times.
- Excluding the work inside the **while** loop, the total work performed by the **for** loop is $O(n)$.
- Let t_i be number of iterations of the **while loop** at the iteration i of the **for** loop.
- The total work inside the **while** loop is $O(t_i)$.

$\Rightarrow$ total time for `InsertionSort` is $O\left(n + \sum_{i=1}^{n-1} t_i\right)$.

Analysis of InsertionSort: Best Case

A is sorted, so $t_i = 1$ for $i = 1, 2, \ldots, n-1$.

$$\Rightarrow \text{total time} = O\left(n + \sum_{i=1}^{n-1} 1\right).$$

Now,

$$n + \sum_{i=1}^{n-1} 1 \quad = \quad n + n - 1 \quad = \quad O(n).$$

$$\Rightarrow \text{total time} = O(n).$$

InsertionSort is linear in best case.

Analysis of `InsertionSort`: Worst Case

A is in reverse order, so $t_i = i$ for $i = 1, 2, \ldots, n-1$.

$$\Rightarrow \text{total time} = O\left(n + \sum_{i=1}^{n-1} i\right).$$

Now,

$$n + \sum_{i=1}^{n-1} i = n + \frac{(n-1)n}{2} = \frac{n^2+n}{2} = O(n^2).$$

$$\Rightarrow \quad \text{total time} = O(n^2).$$

`InsertionSort` is quadratic in worst case.

Binary Search

```
BinarySearch(A, v)
    Input: A sorted array A and a value v.
    Output: true if there is an index i such that A[i] == v; false if
            no such index exists.
    n = A.length
    left = 0
    right = n - 1
    while left ≤ right do
        mid = (left + right)/2
        if A[mid] == v then
            return true
        if v < A[mid] then
            right = mid - 1
        else
            left = mid + 1
    return false
```

Logarithms

Definition

Suppose $b > 1$. The logarithm base b of x is the number y such that

$$\log_b(x) = y \quad \text{exactly if} \quad b^y = x.$$

Examples

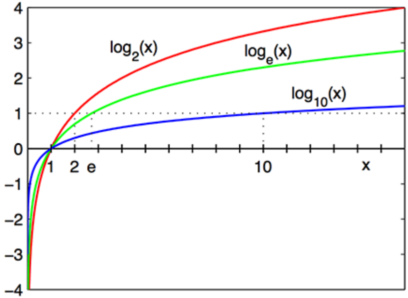
- $\log_2 64 = 6$, since $2^6 = 64$.
- $\log_3 81 = 4$, since $3^4 = 81$.
- $\log_5 32 = \frac{\log_2 32}{\log_2 5} \approx 2.1539$.

Logarithms

Fact

For $b, c > 1$,

$$\log_b x = \frac{\log_c x}{\log_c b}.$$



Source: Wikipedia

Convention

If we do not specify the base, we assume base 2: $\log x$ means $\log_2 x$.

Analysis of Binary Search

Theorem

The worst-case running time of `BinarySearch` *on an n-element array is* $O(\log n)$.

Proof.

- The body of the loop only takes $O(1)$ time, and all steps outside the loop take O(1) time.

  $O(1)$ means that time is bounded by a constant.
- Running time $=$ $\#$iterations $\times O(1) + O(1) = O(\#$iterations$)$.
- $\#$iterations $\leq \log n$.

  Proved next.

□

Analysis of Binary Search

Lemma

The worst-case number of iterations that `BinarySearch` *performs on an* n*-element array is* $O(\log n)$.

Proof.

- Each iteration divides the search range $[\texttt{left} \ldots \texttt{right}]$ by 2.
- The loop terminates when either
  1. we find v ($A[\texttt{mid}] == v$) or
  2. there are no more elements in the search range ($\texttt{left} > \texttt{right}$).
- Let $k = \#$iterations.
- Then, $k \leq$ max number of times we can divide n by 2 before we get 1.
- That is, $n/2^k \geq 1$ or, equivalently, $2^k \leq n$.
- Thus $k \leq \log n$.

□

Logarithmic Running Time

Fact

If the problem size decreases by a constant factor at each iteration, then the number of iterations is a logarithmic function.

Example

Assume n is a positive integer.

$$\textbf{while } (n > 1) \; \{n = (n * 9)/10\}$$

- The loop iterates $\log_{10/9} n = O(\log_2 n)$ times.
- The constant inside the big-O is (≈ 6.59).

Bibliography

References

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms* (3rd edition), MIT Press, 2009.

[KT] Jon Kleinberg and Éva Tardos, *Algorithm Design*, Addison-Wesley, 2006.