

Computing Time – Time Complexities

Textbook Reading:

- Section 2.5, pp. 32-35
- Subsections 2.6.1 & 2.6.2, pp. 36-38

Analyzing Algorithm Performance

- We measure the computing time or (time) **complexity** of an algorithm as a function of the **input size n** to the algorithm.
- For example, when searching or sorting a list, the input size is the number of elements n in the list;
- when evaluating a polynomial, the input size is either the degree of the polynomial or the number of nonzero coefficients in the polynomial;
- when multiplying two square $n \times n$ matrices, the input size is n ;
- when testing whether an integer is a prime, the input size is the number of digits of the integer;
- when traversing a tree, the input size is the number of nodes in the tree; and so forth.

Measuring complexity

- In measuring the complexity (computing time) of an algorithm we identify a **basic operation**.
- And **count how many times an algorithm performs this basic operation**.
- Analysis based on a suitably chosen basic operation yields measurements that are proportional to actual run time behavior exhibited when running the algorithm on various computers, so that the analysis is not dependent on a particular computer.

Catch



- Counting the number of basic operations is not well-defined, even restricted to a given input size n , because a different number of operations may be performed for different inputs of the same size.
- This is solved by defining best-case, worst-case and average complexities.

Best-Case Complexity

The *best-case complexity* $B(n)$ of an algorithm is the **fewest** basic operations performed over all inputs of **size n** .

This can be expressed mathematically as follows:

$$B(n) = \min\{ \tau(I) \mid I \text{ in } \mathcal{I}_n \}$$

Worst-Case Complexity

The *worst-case complexity* $W(n)$ of an algorithm is the **most** basic operations performed over all inputs of **size** n .

This can be expressed mathematically as follows:

$$W(n) = \max \{ \tau(I) \mid I \text{ in } \mathcal{I}_n \}$$

Average Complexity

We define a random variable τ that maps the sample space \mathcal{I}_n of all inputs I of **size n** onto the number of basic operations performed by the algorithm for input I . The average complexity $A(n)$ is defined to be the **expected value** of τ , i.e.,

$$A(n) = E(\tau).$$

Note that the average complexity $A(n)$ is **dependent on the probability distribution** on \mathcal{I}_n .

A review of probability theory and random variables is given in Appendix E of the textbook *Algorithms: Foundations and Design Strategies*.

Formula Average Complexity

Let p_i denote the number of basic operations algorithm performs for an input of size n .

Then

$$A(n) = \sum_{i=B(n)}^{W(n)} ip_i$$

Linear Search

function *LinearSearch* ($L[0:n-1], X$)

Input: $L[0:n-1]$ (a list of size n), X (a search item)

Output: returns index of first occurrence of X in the list, or -1 if X is not in the list

for $i \leftarrow 0$ **to** $n-1$ **do**

if ($X = L[i]$) **then**

return(i)

endif

endfor

return(-1)

end *LinearSearch*

Complexity Analysis of *LinearSearch*

The basic operation of *LinearSearch* is the **comparison** of the search element to a list element. In the input size n is the size of the list.

Best-Case Complexity

Clearly, *LinearSearch* performs only one comparison when the input X is the first element in the list, so that the best-case complexity is

$$B(n) = 1.$$

Worst-case complexity

The most comparisons are performed when X is not in the list, or when X occurs in the last position only. Thus, the worst-case complexity of *LinearSearch* is

$$W(n) = n.$$

Average Complexity of *LinearSearch*

To simplify the discussion of the average behavior of *LinearSearch*, we assume that the search element X is in the list $L[0:n - 1]$ and is equally likely to be found in any of the n positions. Note that i comparisons are performed when X is found at position i in the list. Thus, the probability that *LinearSearch* performs i comparisons is given by $p_i = 1/n$.

Substituting these probabilities into the formula for $A(n)$ yields:

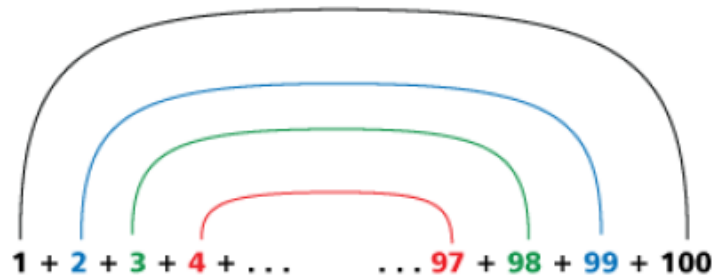
$$A(n) = \sum_{i=B(n)}^{W(n)} ip_i = \sum_{i=1}^n i \frac{1}{n} = \frac{1}{n} (1 + 2 + \dots + n).$$

Can we obtain a closed-form formula for
 $1 + 2 + \dots + n$?

Sum of numbers $1 + 2 + \dots + 100$



I love the story of Carl Friedrich Gauss—who, as an elementary student in the late 1700s, amazed his teacher with how quickly he found the sum of the integers from 1 to 100 to be 5,050. Gauss recognized he had fifty pairs of numbers when he added the first and last number in the series, the second and second-last number in the series, and so on. For example: $(1 + 100)$, $(2 + 99)$, $(3 + 98)$, \dots , and each pair has a sum of 101.



$50 \text{ pairs} \times 101 \text{ (the sum of each pair)} = 5,050$.

Another way to represent the problem could be to list the integers from 1 to 100 and write the same list in reverse

$$1 + 2 + 3 + 4 + \dots + 97 + 98 + 99 + 100$$

order below the first list. $100 + 99 + 98 + 97 + \dots + 4 + 3 + 2 + 1$

$$101 + 101 + 101 + 101 + \dots + 101 + 101 + 101 + 101$$

This gives us 100 addends of 101 for 10,100. Because the list of numbers from 1 to 100 was doubled, we need to divide the total by 2, giving us a sum of 5,050.

Formula for $1 + 2 + \dots + n$

Let $S = 1 + 2 + \dots + n$.

$$1 + n = n + 1$$

$$2 + n - 1 = n + 1$$

$$3 + n - 2 = n + 1$$

\vdots

$$\frac{n + 1 = n + 1}{S + S = n(n + 1)}$$

$$S + S = n(n + 1)$$

Summing columns we obtain

$$2S = n(n+1) \text{ or } S = n(n+1)/2$$

Substituting formula for $1 + 2 + \dots + n$ we obtain:

$$A(n) = \frac{1}{n} (1 + 2 + \dots + n) = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

Binary Search

Now suppose as a precondition the list is sorted. Then, there is a much faster algorithm for searching called Binary Search.

Pseudocode for Binary Search

function *BinarySearch* ($L[0:n-1], X$)

Input: $L[0:n-1]$ (a sorted array of n list elements)
 X (a search item)

Output: returns the index of an occurrence of X in the list, or -1 if X is not in the list

$Found \leftarrow \text{.false.}$

$low \leftarrow 0$

$high \leftarrow n - 1$

while **.not.** $Found$ **.and.** $low \leq high$ **do**

$mid \leftarrow \lfloor (low + high)/2 \rfloor$

if $X = L[mid]$ **then** $Found \leftarrow \text{.true.}$

else if $X < L[mid]$ **then**

$high \leftarrow mid - 1$

else

$low \leftarrow mid + 1$

endif

endif

endwhile

if $Found$ **then**

return(mid)

else

return(-1)

endif

end *BinarySearch*

Recursive Version

function *BinarySearch*($L[0:n-1]$, low, high, X)

Input: $L[0:n-1]$ (an array of n list elements, sorted in increasing order)
 low, high (nonnegative integers)
 X (a search item)

Output: returns the index of an occurrence of X in the sublist $L[0:n-1, \text{low}, \text{high}]$ or -1 if X is not in the list

if $\text{high} < \text{low}$ **then return**(-1) **endif** // empty list

$\text{mid} \leftarrow \lfloor (\text{low} + \text{high})/2 \rfloor$

if $X = L[\text{mid}]$ **then return**(mid) **endif**

if $X < L[\text{mid}]$ **then**

BinarySearch($L[0:n-1]$, low, $\text{mid} - 1$, X)

else

BinarySearch($L[0:n-1]$, $\text{mid} + 1$, high, X)

endif

end *BinarySearch*

PSN. Write in C++. Include good commenting.

Best-case complexity

The best-case complexity of *BinarySearch* occurs when X is found in the midpoint position $\lfloor (n - 1)/2 \rfloor$ of $L[0:n - 1]$ and involves a single operations, i.e.,

$$B(n) = 1$$

Worst-case complexity

The worst-case complexity is equal to twice the longest string of midpoints (values of *mid*) ever generated by the algorithm for an input X . In particular, if we assume that $n = 2^k - 1$ for some positive integer k , then such a string is generated by searching for $X = L[0]$. We then compare X successively to the midpoints $2^{k-1} - 1, 2^{k-2} - 1, \dots, 0$, so that this longest string has length k . To express k in terms of n , we note that $n + 1 = 2^k$, so that we have $k = \log_2(n + 1)$. Since two comparisons are involved at each stage

$$W(n) = 2 \log_2(n + 1) \approx 2 \log_2 n.$$

PSN. Give C++ code for the a recursive version of binary search where no equality check with midpoint is made and analyze your algorithm.

Complexity Analysis

For convenience assume $n = 2^k$. Then

$$B(n) = W(n) = k = \log_2 n.$$

For any probability distribution on the sample space of inputs of size n

$$B(n) \leq A(n) \leq W(n).$$

It follows that

$$A(n) = \log_2 n.$$

Why did the computer show up at work late?



Answer:

It had a hard drive.

