Algorithm Analysis

Algorithm

An algorithm is a finite set of well-defined instructions that takes some set of values as *input* and produces some set of values as *output* to solve a problem. An algorithm is thus a sequence of computational steps that transform the input into the output.

Computational Complexity

Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to demonstrate that your solution method terminates and does so with the correct answer.

If computers were infinitely fast, any correct method for solving a problem would do. You would probably want your implementation to be within the bounds of good software engineering practice, but you would most often use whichever method was the easiest to implement.

Of course, computers cannot be infinitely fast, and memory cannot be free. Computing time is therefore a bounded resource, and so is space in memory. These resources should be used wisely, and algorithms that are efficient in terms of time or space will help you do so. Therefore, given several algorithms to solve a problem, we want to determine the amount of memory they use and how much time they require, so that we can choose the most efficient algorithm among them. This is called an *algorithm analysis*. So we introduce the term *complexity*, or efficiency to analyze algorithms. And there are two major complexities. *Time complexity* and *space complexity*, which are respectively how many steps does it take to perform a computation, and how much memory is required to perform that computation.

Analyzing an algorithm: Maximum Contiguous Subsequence Sum Problem

Analyzing an algorithm has come to mean predicting the resources (time, memory and etc.) that the algorithm requires. And most often it its computation time that we want to measure. Here, we give several algorithms to solve the problem called *Maximum Contiguous Subsequence Sum* and analyze them to determine which algorithm is efficient, or which algorithm has the smallest time complexity.

The problem is defined as following:

> *Maximum Contiguous Subsequence Sum* problem:
> Given an array with integers $a_1, a_2, \ldots, a_n$, find the maximum value of $\sum_{k=i}^{j} a_k$, over all possible choices of i and j.

For instance, if the input is $\{-2, \mathbf{11, -4, 13}, -5, 2\}$, then the answer is 20, which represents the contiguous subsequence encompassing items $11, -4, 13$ (shown in boldface type).

The time taken by an algorithm solving this problem depends on the size of the input: finding the maximum value in sequence of thousand numbers takes longer than finding the answer for five numbers. In general, the time taken by an algorithm grows with the size of the input.

The naive algorithm

There is an obvious solution to the problem: Just compute the sum for each possible subsequence, and take the largest one. We first note that the number of subsequences is $\binom{n}{2} + n = \frac{n^2}{2} + \frac{n}{2}$.

```
def maxSubSum1(a):
  maxSum = 0
  start = 0
  end = 0
  for i in range(len(a)):
    for j in range(i, len(a)):
```

```
      sum = 0
      for k in range(i, j+1):
        sum += a[k]
      if sum > maxSum:
        maxSum = sum
        start = i
        end   = j
  return maxSum, start, end
```

Analysis. We will only count the number of additions performed by the algorithm, that is, how often the line

```
sum += a[k]
```

is executed. The number of additions is

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1) = \sum_{i=0}^{n-1}(1+2+\cdots+(n-i)) = \sum_{i=0}^{n-1}\frac{(n-i)(n-i+1)}{2} = \sum_{k=1}^{n}\frac{k(k+1)}{2}$$
$$= \frac{1}{2}(\sum_{k=1}^{n}k^2 + \sum_{k=1}^{n}k) = \frac{n(n+1)(2n+1)}{12} + \frac{n(n+1)}{4} = \frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}.$$

The faster algorithm

If we think about it for a moment, we realize that the sum for the subsequence from i to j can be obtained from the sum for the subsequence from i to $j-1$ with only one addition, namely $\sum_{k=i}^{j} a_k = a_j + \sum_{k=i}^{j-1} a_k$. We can therefore removing the innermost loop and obtain the following code.

```
def maxSubSum2(a):
  maxSum = 0
  start = 0
  end = 0
  for i in range(len(a)):
    sum = 0
    for j in range(i, len(a)):
      sum += a[j]
      if sum > maxSum:
        maxSum = sum
        start = i
        end   = j
  return maxSum, start, end
```

Analysis. Again we only count additions, and obtain the result

$$\sum_{i=0}^{n-1}(n-i) = n + (n-1) + \cdots + 1 = \sum_{i=1}^{n} k = \frac{n^2}{2} + \frac{n}{2}.$$

We note that this is exactly the number of possible subsequences, so this seems already quite good: we only look at each subsequence once.

A recursive algorithm

But it turns out we can do better. We use a standard recursive algorithm design technique called *divide and conquer*. In general, divide and conquer consists of the following three steps:

1. Split the problem into smaller instances of the same problem.
2. Solve the subproblems recursively.
3. Combine the solutions to the subproblems to solve the original problem.

We apply this to the Maximum Contiguous Subsequence Sum problem: We split the original sequence in the middle, and obtain two smaller sequences of half the size. We note that one of the following three cases must hold:

- The maximal subsequence is in the left half.
- The maximal subsequence is in the right half.
- The maximal subsequence begins in the left half and ends in the right half.

We implement this as follows:

```
def maxSumRec(a, left, right):
  if left == right:  # base case
    if a[left] > 0:
      return a[left]
    else:
      return 0
  else:
    center = (left + right) // 2
    maxLeftSum  = maxSumRec(a, left, center)
    maxRightSum = maxSumRec(a, center + 1, right)

    maxLeftBorderSum = 0
    maxRightBorderSum = 0
    leftBorderSum = 0
    rightBorderSum = 0

    for i in range(center, left-1, -1):
      leftBorderSum += a[i]
      if leftBorderSum > maxLeftBorderSum:
        maxLeftBorderSum = leftBorderSum
    for i in range(center + 1, right+1):
      rightBorderSum += a[i]
      if rightBorderSum > maxRightBorderSum:
        maxRightBorderSum = rightBorderSum
    return max(maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum)
```

Analysis. The key insight is that it takes only $\frac{n}{2} + \frac{n}{2} = n$ additions to compute the maximal subsequence that begins in the left half and ends in the right half.

So, let us define $T(n)$ to be the number of additions for n elements. Then,

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(\frac{n}{2}) + T(\frac{n}{2}) + n = 2T(\frac{n}{2}) + n & \text{otherwise} \end{cases}$$

We claim that $T(2^k) = k2^k$. We prove this by induction:

1. Base case: If $k = 0$ then $T(2^0) = T(1) = 0 = 0 \cdot 2^0$. $\surd$
2. Inductive step: If $k > 0$ we assume that $T(2^{k-1}) = (k-1) \cdot 2^{k-1}$ holds.
   Then, $T(2^k) = 2T(2^{k-1}) + 2^k = 2(k-1) \cdot 2^{k-1} + 2^k = k \cdot 2^k$. $\surd$

If we express this again in terms of the input size n, we get $T(n) = n \log n$.

Comparing the algorithms

We implement and run all the algorithms. For $n \geqslant 100$ the recursive algorithm is fastest.

This is not surprising. For $n = 10^6$, for instance, the first algorithm needs roughly $3 \cdot 10^{17}$ additions, the second one $5 \cdot 10^{11}$, and the recursive algorithm less than $2 \cdot 10^7$. If we assume that our processor can do 10^9 additions per second (one addition per cycle on a 1 GHz processor), that means that the first algorithm needs $3 \cdot 10^8$ seconds (about 10 years), the second one 500 seconds (about 8 minutes), and the recursive one only 20 milliseconds.

Interestingly, for $n = 10$ the second algorithm is fastest, even though it performs 55 additions, while the recursive algorithm only needs about 30 additions. The reason the recursive algorithm is a bit slower is that it contains more overhead in the form of recursive function calls.

Can we do better?

It is in fact possible to solve the problem even faster, with a number of additions linear in n. We leave this as an exercise to the reader. `:-)`

Analyzing an algorithm

To compare which algorithm is faster, one method is to write a program implementing the algorithm and run on real machines.

But this measurement have several limitations. To use this measurement we should implement the algorithm. Implementation may be time-consuming or difficult, and the algorithm may have undiscovered drawbacks for untested inputs. If we are going to compare two algorithms, we should use the same hardware and software environment. If we are going to compare a new algorithm with an old algorithm, we need to install the same environment where the old algorithm was tested, or reimplement the old algorithm and run in the new computer environment. To compare several algorithms, we need to and run them all. Finally, we need to make sure that we have test cases that are representative of the algorithms behavior.

Theoretical analysis

The idea of *theoretical analysis* is to count the *primitive operations* that an algorithm executes in the worst case on an input of size n. Examples of primitive operations are:

- Assigning a value to a variable
- Calling a method
- Arithmetic operations (e.g. adding two numbers)
- Indexing into an array
- Following a reference
- Returning from a method

For example, assume that there are n elements in a linked list, and we execute this function to compute the last element in the list `L`:

```
def last(L):
  p = L._front
  while p.next is not None:
    p = p.next
  return p
```

The first line is one primitive operation (an assignment), the last line is one primitive operations (returning from a method). The loop body (line 4) consists of two primitive operations, and is executed $n-1$ times. The loop condition is two primitive operations, and is executed n times. In total, we have counted $4n$ primitive operations.

All primitive operations take constant time on modern computers. Depending on the speed of the processor, on the processor and computer architecture, and the current caching status, this may be 10 nanoseconds or several hundreds of nanoseconds, but the difference is a reasonably small factor. So counting primitive operations is helpful to estimate the running time. Let's assume that the fastest primitive operation takes time a and the slowest operation takes time b. If the algorithm uses $P(n)$ primitive operations, then its running time $T(n)$ is bounded by:

$$aP(n) \leqslant T(n) \leqslant bP(n)$$

Growth rate

In fact, instead of the specific funning time for a given size of problem, we are actually more interested in the *growth rate* of the running time. The growth rate indicates how fast the running time increases when the size of input increases.

Why is this important? Because the growth rate determines the scaling behavior. The scaling behavior is how the running time changes when the size of input multiplies by some factor. Inversely, it is related to the size of input we can solve in a time interval when performance of the computer increases by some factor.

There are problems with great amount of input size. 3D graphics is an example. We should handle hundred million triangles to draw. If the scaling factor of the algorithm is bad, although computing speed goes faster, only little size of input would increases. Although you may think that algorithms are not important because computing speed increases rapidly, algorithm is even more important to increase the input size to be processed.

Time complexity	Problem size after speedup
n	$10s$
n^2	$3.16s$
n^3	$2.15s$
2^n	$s + 3.3$

Table 1: solvable problem size after 10-times speedup

The table above shows how much larger the input we can solve in the same time becomes if the computer becomes 10 times faster. If the time complexity is linear, we can solve a 10-times-larger problem. If the complexity is exponential, for instance 2^n, then the size of problem we can solve does not grow by some factor. Only problems with size $s + \log(10) \simeq s + 3.3$ can be solved, where s is the size of the original problem.

Big-Oh notation

Big-Oh notation represents the growth rate of an algorithm. Big-Oh notation shows the comparison result of the growth rate of two function.

Definition Let $f(n)$, $g(n)$ be functions from $\{1, 2, 3, 4, ...\}$ to $\mathbb{R}$. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer $n_0 \geqslant 1$ such that

$$f(n) \leqslant cg(n) \qquad \text{for } n \geqslant n_0.$$

For example, since $4n+1 \leqslant 5n$ for $n \geqslant 1$, $4n+1 = O(n)$. This is not the only way to show that $4n+1 = O(n)$. Since $4n + 1 \leqslant 4.1n$ for $n \geqslant 10$, $4n + 1 = O(n)$.

Let's prove that $2n^2+3n+5 = O(n^2)$. Since $n \leqslant n^2$ for $n \geqslant 1$ and $1 \leqslant n^2$ for $n \geqslant 1$, $2n^2+3n+5 \leqslant 10n^2$ for $n \geqslant 1$. So $2n^2 + 3n + 5 = O(n^2)$.

Since $2^{n+2} = 4 \cdot 2^n \leqslant 5 \cdot 2^n$ for $n \geqslant 1$, $2^{n+2} = O(2^n)$.

Simplest terms

We want to express the running time in the simplest possible Big-Oh notation. If $f(n) = O(g(n))$, $f(n) = O(h(n))$, you may want to describe the running time in the simpler form between $g(n)$ and $h(n)$, if $g(n) = O(h(n))$ and $h(n) = O(g(n))$.

For any polynomial

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d$$

with $a_d > 0$, $f(n)$ is just $O(n^d)$.

Note that $\log n \leqslant n$ for $n \geqslant 0$, so $n \log n = O(n^2)$.

time complexity	the simplest term
$5n^2 + 3n \log n + 2n + 5$	$O(n^2)$
$20n^3 + 10n \log n + 5$	$O(n^3)$
$3 \log n + 2$	$O(\log n)$
2^{n+2}	$O(2^n)$
$2n + 100 \log n$	$O(n)$

Table 2: time complexities and their simplest forms

A drawback of asymptotic analysis

Even though an algorithm with lower growth rate is faster than an algorithm with higher growth rate when input size is very large, we cannot say that the algorithm with lower growth rate is always better. Here is an extreme and unrealistic example: Assume we have three algorithms with running times $10^{100}n$, n^{100}, and 2^n for input size n. When n is very large, the first one is the best, the third one is the worst. But such values of n are completely unrealistic—there are not enough atoms in the universe to represent so much data—and the exponential algorithm is actually the best algorithm in practice.

Time complexity on several data structures

Since this is a data structure course, we will use algorithms analysis to analysis operations on data structures.

When we look at an abstract data type (ADT), it has several operations (i.e. searching, inserting, removing, $\cdots$). Some operations may be used very frequently, and some operations may be used very infrequently. We often make a trade-off: we optimize only the frequently used operations.

The `index(x)` method of Python lists returns the first element in the list equal to `x`. How would we implement this method?

Here is the obvious implementation:

```
def index(self, x):
  for i in range(len(self)):
    if self[i] == x:
      return i
  return -1
```

The input size n here is the size of the list. In the worst case, we will not find the object. Since we have to look at the entire list before we can return `-1`, the running time is $O(n)$. In the best case, the item would be in front of the list and it would be found in just one iteration. In this case the running time becomes $O(1)$.

Can we say something about the average case? If the element is not on the list, the method takes time $O(n)$, so this is not very interesting. What if we take the average over all possible elements that are on the list? The average number of list positions examined is then

$$\frac{1}{n}\sum_{i=0}^{n-1} i = \frac{1}{n} \cdot \frac{n(n-1)}{2} = \frac{n-1}{2} = O(n).$$

Average-time analysis is generally difficult. One problem is to decide on what is the right probability model for the input.