Algorithms and Their Running Times

Introduction

The development of data structures and algorithms is motivated by the need for computer programs to run on machines as quickly as possible, for inputs of all sizes, and using as little memory as possible (although in general minimizing program speed and memory usage are two competing objectives). With this in mind, for each algorithm, program function, or data-structure operation we associate with it a **running time** $T(n)$, where n is a parameter that reflects the size of the problem input (in the case of algorithm or function execution) or the size of the data-structure when the data-structure operation is performed. Moreover $T(n)$ represents the worst-case running time for all inputs whose size can be represented by n. For example, if the algorithm sorts arrays integers, then we may let n denote the length of the array being sorted. Sometimes we need more than just one parameter to represent the size of the input. For example, an algorithm that takes graphs as input, will have a running time of the form $T(m, n)$ where m is the number of graph edges, and n the number of vertices of the problem.

1 Using Big-O Notation to Describe Running Time

When communicating a running time $T(n)$, we must decide on how to represent it with big-O notation. In other words, when do we use big-O? big-Ω?, big-Θ? Although $T(n)$ refers to the worst-case running time for an input of size n, the choice of big-O notation can in addition provide information about the **best-case** running time $T_b(n)$.

- $T(n) = O(f(n))$ means that the worse-case running time is achieved by $f(n)$, but $T_b(n) = o(f(n))$. In other words, $T(n)$ and $T_b(n)$ have different orders of growth.
- $T(n) = \Theta(f(n))$ means that both T_b and T have the same order of growth as $f(n)$.
- $T(n) = \Omega(f(n))$ means that there are inputs of size n that force the running time to equal or exceed $f(n)$.

Example 1. a) Use big-O notation to provide the running time $T(n)$ for the binary search algorithm which takes as input an array a of size n and an element x, and returns the location of x in a if it exists, and returns -1 otherwise. b) Same question, but now $T(n)$ is the time needed to convert nonnegative decimal integer n to a binaray number.

Why use big-O notation? Because it makes no sense to give an exact physical running time (say in seconds) for an algorithm, since this time will be highly dependent on the following factors.

- the programming language that implements the algorithm;
- the compiler or interpreter of the programming language that creates the machine program;
- the speed and memory specifications of the computer that runs the program;
- the number of processors available to the computer that runs the program;
- the computer's operating system;
- the other processes running on the computer;
- network speed in case the algorithm is distributed over a network of processors.

All of the above factors will affect the running time by some constant factor. For this reason, constants are ingnored in the analysis of running time, and hence big-O notation represents the ideal notation for describing $T(n)$.

Analyzing the Running Time of Loops

As programmers we want to keep the running time of our programs as low as possible. High running times (e.g. quadratic, cubic, and higher) often arise from the (over) use of loops. In this section, the goal is to become proficient at modeling the running time of a loop (or nested loop) with a summation expression. The expression is then evaluated to determine the (big-O) running time.

The following formulas will be helpful when evaluating summation expressions.

- $\sum_{i=1}^{n} 1 = n$.
- $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$.
- $\sum_{i=1}^{n} i^2 = 1 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$.
- $\sum_{i=1}^{n} i^3 = 1 + 2^3 + \cdots + n^3 = [\frac{n(n+1)}{2}]^2$.
- $\sum_{i=0}^{n} ar^i = a + ar + ar^2 + \cdots + ar^n = \frac{a(r^{n+1}-1)}{r-1}$.
- $\sum_i (af(i) + bg(i) = a\sum_i f(i) + b\sum_i g(i)$, where a and b are constants, and f and g are functions that depend on i.

Example 2. Evaluate the summation expression a) $\sum_{i=1}^{n}(7i^2 + i + 8)$, and b) $\sum_{i=1}^{n}\sum_{j=1}^{i}(3i + 2j)$.

Use the following guidelines for determining the big-O running time of one of the code snippets in this lecture.

1. Assume the size parameter is equal to n.
2. The number of nested summations should equal the number of nested loops.
3. The summation expression should omit constants.
4. The summation expression should be evaluated *exactly* until it is has been transformed into an expression that only depends on n. This expression, in turn, can be replaced with a succinct big-O or big-Θ expression.

Example 3. Let $T(n)$ be the worst-case running time for the following piece of code. Model $T(n)$ with a summation expression. Evaluate the expression in order to obtain the order of growth of $T(n)$.

```
int linear_search(int x, int a[], int n)
{
    for(i=1; i <= n; i++)
        if(a[i] == x)
            return true;

    return false;
}
```

Solution to Example 3. Since the above code only has one loop, and the number of steps per loop is O(1), the desired summation is

$$\sum_{i=1}^{n} 1 = \Theta(n),$$

and the code executes in linear time.

Example 4. Let $T(n)$ be the worst-case running time for the following piece of code. Model $T(n)$ with a summation expression. Evaluate the expression in order to obtain the order of growth of $T(n)$.

```
int binary_search(int x, int a[], int n)
{
    int left = 0;
    int right = n-1;
    int mid, diff;

    while(right-left > 1)
    {
        mid = (left+right)/2;
        diff = x-a[mid];

        if(diff == 0)
            return mid;
        else if(diff > 0)
            left = mid;
        else right = mid;
    }

    if(a[left] == x) return left;
    if(a[right] == x) return right;
    return -1; //x is not in the array
}
```

Solution to Example 4. Since the above code only has one loop, and the number of steps per loop is O(1), the desired summation is

$$\sum_{i=1}^{\log n} 1 = \mathrm{O}(\log n),$$

and the code executes in logarithmic time. Notice that i) the summation iterates only $\log n$ times, since the difference between left and right is being halved in each iteration, and ii) we use big-O instead of big-Θ, because the best case running time is $\Theta(1)$.

Example 5. Suppose that $f(n)$ has a runnning time of $\Theta(n)$ while $g(n)$ has a running time of $\Theta(n^2)$. Let $T(n)$ be the worst-case running time for the following piece of code. Model $T(n)$ with a summation expression. Evaluate the expression in order to obtain the order of growth of $T(n)$.

```
sum=0;

for(i=0; i< f(n); i++)
     sum += g(i)
```

Example 6. Let $T(n)$ be the worst-case running time for the following piece of code. Model $T(n)$ with a summation expression. Evaluate the expression in order to obtain the order of growth of $T(n)$.

```
sum=0;
for(i=0;i<n;i++)
    for(j=0; j< n*n;j++)
        sum++;
```

Example 7. Let $T(n)$ be the worst-case running time for the following piece of code. Model $T(n)$ with a summation expression. Evaluate the expression in order to obtain the order of growth of $T(n)$.

```
sum=0;

for(i=0;i<n;i++)
    for(j=0; j< i;j++)
        sum++;
```

The following examples suggest that asymptotic differences in algorithm running times can have a profound impact on the maximum size of a problem that can be computed within a given amount of time.

Example 8. Suppose a machine on the average takes 10^{-8} seconds to execute a single algorithm step. What is the largest input size for which the machine will execute the algorithm in 2 seconds, assuming the number of steps of the algorithm is $T(n) =$

1. $\log n$
2. $\sqrt{n}$
3. n
4. $n \log n$
5. n^2
6. n^3
7. 2^n?

Example 9. For the machine in the previous example, how long will it take the machine to execute the algorithm for an input of size 1,000, assuming the time complexities from the same example?

Case Study: Finding Maximum Subsequence Sum

Maximum Subsequence Sum Problem: given integers $a_1, a_2, \ldots, a_n$, find the maximum value of $\sum_{k=i}^{j} a_k$.

Example 10. Find the maximum subsequence sum for the integers $-2, 11, -4, 13, -5, -2$.

Freshman Algorithm for Maximum Subsequence Sum Problem

.

Assume the integers are $a_1, a_2, \ldots, a_n$

```
max_sum = 0;

for(i=0; i< n; i++)
{
    for(j=i; j< n; j++)
    {
        this_sum=0;

        for(k=i; k <= j; k++)
            this_sum += a[k];

        if(this_sum > max_sum)
            max_sum = this_sum;
    }
}
return max_sum;
```

Running-Time Analysis of Freshman Algorithm:

Sophomore Algorithm for Maximum Subsequence Sum Problem

```
max_sum = 0;

for(i=0; i< n; i++)
{
    this_sum= 0;

    for(j=i; j< n; j++)
    {

        this_sum += a[j];

        if(this_sum > max_sum)
            max_sum = this_sum;
    }
}
return max_sum;
```

Running-Time Analysis of Sophomore Algorithm:

Junior Algorithm for Maximum Subsequence Sum Problem

The junior student has taken 328, and understands the meaning of *"divide and conquer"*. The junior algorithm thus has the strategy of finding the MSS for the first half of the sequence, the MSS for the second half, the MSS that intersects both halves, and takes the largest of the three.

```
//Assume: we are only interested in the MSS that is found between a[left] and a[right].
//Initial call to mss_junior: mss_junior(a,0,n-1)
int mss_junior(int[ ] a, int left, int right)
{
    //Base case 1
    if(right == left)
        return a[left];

    //Base case 2
    if(right == left+1)
        return max(a[left],a[right],a[left]+a[right])

    int mid = (left+right)/2;

    //Find the MSS that occurs in the left half of a
    int mss_left = mss_junior(a,left,mid);

    //Find the MSS that occurs in the right half of a
    int mss_right = mss_right(a,mid+1,right);

    //Find the MSS that intersects both the left and right halves
    //EXERCISE: implement mss_junior_middle()
    int mss_middle = mss_junior_middle(a,left,mid,right);

    return max(mss_left,mss_right,mss_middle)
}
```

Running-Time Analysis of Junior Algorithm:

Senior Algorithm for Maximum Subsequence Sum Problem

```
max_sum = 0;
this_sum = 0;

for(i=0; i< n; i++)
{
    this_sum += a[i];

    if(this_sum > maxSum)
        max_sum = this_sum;
    else if(this_sum < 0)
        this_sum = 0;
}
return max_sum;
```

Big-O Analysis of Senior Algorithm:

Exercises

1. Simplify each summation to an expression in terms of n, and provide the big-Θ growth of the expression.

   (a) $\sum_{i=1}^{n}(n-2i+3)$
   (b) $\sum_{i=0}^{n-1}(4i^2-2i+7)$
   (c) $\sum_{j=10}^{n} j$
   (d) $\sum_{i=1}^{n}\sum_{j=1}^{n} j$
   (e) $\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i)$

2. For each of the following code fragments, provide an appropriate summation expression that models its running time $T(n)$. Then simplify the summation expression to an expression in terms of n, and then determine either a big-O or big-Θ representation of running time $T(n)$.

```
a.
sum = 0;
for(i=0; i < n; i++)
    sum++;

b.
sum = 0;
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        sum++;

c.
sum = 0;
for(i=0; i < n; i++)
    for(j=0; j < n*n; j++)
        sum++;

d.
sum = 0;
for(i=0; i < n; i++)
    for(j=0; j < i; j++)
        sum++;
```

```
e.
sum = 0;
for(i=0; i < n; i++)
    for(j=0; j < i*i; j++)
        for(k=0; k < j; k++)
            sum++;
```

```
f.
sum = 0;
for(i=1; i <= n; i++)
    for(j=1; j <= i*i; j++)
        if(j % i == 0)
            for(k=0; k < j; k++)
                sum++;
```

3. The function

   `int [ ] add(int[ ] a, int [ ] b, int length)`

   inputs two length-n arrays a and b whose elements represent the digits of two nonnegative integers that are to be added (using the algorithm learned in elementary school). Here we assume that $a[0]$ and $b[0]$ hold the least-signifanct digits of the two integers . The function then returns an array that holds the digits of $a + b$. For example, to add 472 to 54, we call `add` on arrays $a = 2, 7, 4$ and $b = 4, 5, 0$, with $n = 3$, and the function returns the array $6, 2, 5$. Implement this function using "Java" or "C"-like pseudocode. Then provide an appropriate summation expression that models its running time $T(n)$. Simplify the summation expression to an expression in terms of n, and use it to determine either a big-O or big-Θ representation of running time $T(n)$.

4. The function

   `int [ ] multiply(int[ ] a, int [ ] b, int length)`

   inputs two length-n arrays a and b whose elements represent the digits of two nonnegative integers that are to be multiplied (using the algorithm learned in elementary school). Here we assume that $a[0]$ and $b[0]$ hold the least-signifanct digits of the two integers. The function then returns an array that holds the digits of $a \times b$. For example, to multiply 54 and 145, we would pass in the arrays $a = 4, 5, 0$ and $b = 5, 4, 1$, and the function should return the array $0, 3, 8, 7$. Implement this function using "Java" or "C"-like pseudocode. Then provide an appropriate summation expression that models its running time $T(n)$. Simplify the summation expression to an expression in terms of n, and use it to determine either a big-O or big-Θ representation of running time $T(n)$. Hint: first implement a function that multiplies a nonnegative integer by a single digit, then call this function, along with the `add` function within a loop to complete the entire multiplication.

5. An algorithm takes 0.5 seconds to run on an input of size 100. How long will it take to run on an input of size 1000 if the algorithm has a running time that is linear? quadratic? log-linear? cubic?

6. An algorithm is to be implemented and run on a processor that can execute a single instruction in an average of 10^{-9} seconds. What is the largest problem size that can be solved in one hour by the algorithm on this processor if the number of steps needed to execute the algorithm is n? n^2?, n^3? 1.3^n, $n \log n$? Assume n is the input size.

7. Suppose that the Insertion Sort sorting algorithm has a running time of $T(n) = 8n^2$, while the Counting Sort algorithm has a running time of $T(n) = 64n$. Find the largest positive input size for which Insertion Sort runs at least as fast as Counting Sort.

8. If you were given a full week to run your algorithm, which has running time $T(n) = 5 \cdot 10^{-9}(n^3)$ seconds, what would be the largest input size that could be used, and for which your algorithm would terminate after one week? Explain and show work.

9. Consider the problem of computing a^n where n is a positive integer. One method is to start with a product of 1, iterate n times, and multiply the product by a each time. The following, however, is a faster approach. Write n as a binary number. For example, suppose, $n = 2^{b_1} + \cdots + 2^{b_r}$, then start with a product of 1, and multiply the product by each of $a^{2^{b_i}}$. In other words, we only multiply with exponents that are powers of 2. For example, to compute 3^6, we would multiply 1 by both 3^4 and 3^2, since $3^6 = 3^{2+4}$. How does this reduce the running time? Hint: how many multiplications does this algorithm need? Implement this algorithm in pseudocode.

10. Given as input a sorted integer array $a[0] < a[1] < \cdots < a[n-1]$, provide an algorithm with $O(\log n)$ running time. that checks if there is an i for which $a[i] = i$. Describe your algorithm in words. Then provide supporting pseudocode.

11. Provide a linear-time algorithm for finding the maximum subsequence product for an array of integers. Argue that your algorithm is correct, and runs in linear time. Provide the running time of the algorithm.

12. Describe how you could modify any algorithm so that it has a good (say $O(1)$ or $O(n)$) best-case running time.

Exercise Solutions

1. Note: the final expressions have been simplified (which the exercise did not require).

   (a) $\sum_{i=1}^{n}(n-2i+3) = n^2 - n(n+1) + 3n = 2n = \Theta(n)$.

   (b) $\sum_{i=0}^{n-1}(4i^2-2i+7) = \frac{2(n-1)(n)(2n-1)}{3} - n(n-1) + 7n = \Theta(n^3)$.

   (c) $\sum_{j=10}^{n} j = \frac{n(n+1)}{2} - \frac{9(10)}{2} = \Theta(n^2)$.

   (d) $\sum_{i=1}^{n}\sum_{j=1}^{n} j = \sum_{i=1}^{n}(n^2/2 + n/2) = n^3/2 + n^2/2 = \Theta(n^3)$.

   (e) $\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i) = \sum_{i=1}^{n}(n^2/2 + n/2 - i^2/2 + i/2 - ni + i^2 - i)$. Then evaluate the outer sum to get $\Theta(n^3)$.

2. (a) $\sum_{i=0}^{n-1} 1 = n = \Theta(n)$.

   (b) $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} n = \frac{n(n-1)}{2} = \Theta(n^2)$.

   (c) $\sum_{i=0}^{n-1}\sum_{j=0}^{n^2-1} 1 = \sum_{i=0}^{n-1} n^2 = \frac{(2n-1)n(n-1)}{6} = \Theta(n^3)$.

   (d) $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$.

   (e) $\sum_{i=0}^{n-1}\sum_{j=0}^{i^2-1} j = \sum_{i=0}^{n-1} \frac{i^2(i^2-1)}{2} = \Theta(n^5)$. Notice that I did not get an exact answer for the succinct sum, since the last summation involves i^4, and I have not provided you a formula for $\sum_{i=1}^{n} i^4$. However, using the Integral Theorem, we know the sum should have order of growth $\Theta(n^5)$.

   (f) $\sum_{i=1}^{n}\sum_{j|i,j\le i^2} j + \sum_{i=1}^{n}\sum_{j\nmid i,j\le i^2} 1 =$
   $\sum_{i=1}^{n}(i + 2i + \cdots i^2) + \sum_{i=1}^{n}(i^2 - i) = \Theta(n^4)$. Why?

3. The running-time analysis yields $\Theta(n)$.

4. The running-time analysis yields $\Theta(n^2)$.

5. The input size has grown by a factor of 10. For linear time, the running time will also grow by a factor of 10. For quadratic, it will grow by a factor of $10^2 = 100$, to 50 seconds. For cubic, it will grow by a factor of $10^3 = 1000$, to 500 seconds. For log linear, we can first solve for the running time cofficient C (i.e. $T(n) = Cn\log(n)$). This yields $C = 0.5/(100\log(100)$. Then use this value to compute $T(1000)$.

6. The elapsed time will be $10^{-9}T(n)$ seconds which must not exceed 3600. Thus we must have $T(n) \le (3600)(10^9)$, which implies $n = \lfloor T^{-1}((3600)(10^9))\rfloor$. For example, if $T(n) = n^2$, then $T^{-1}(n) = \sqrt{n}$. In this case $n = \lfloor\sqrt{(3600)(10^9)}\rfloor = 1897366$. In the case when $T(n) = n\log n$, use a graphing calculator to determine the value of n for which $T(n)$ is approximately $(3600)(10^9)$. We must do this because there is no formula for computing T^{-1}.

7. Solve the quadratic equation $8n^2 = 64n$ to get $n = 8$. For $n \ge 9$, Counting Sort will run faster.

8. One week contains $(3600)(24)(7) = 604800$ seconds. Thus, the largest input that can be solved in one week is $n = \lfloor(((604800)(10^9)/5)^{1/3}\rfloor = 49455$. So even though you have an entire week to run the program, the largest problem that can be solved will be in the tens of thousands.

9. To compute a^n, if n has the binary representation $n = 2^{i_1} + 2^{i_2} + \cdots + 2^{i_k}$, where $i_1 > i_2 > \cdots > i_k$, then
$$a^n = a^{2^{i_1}+2^{i_2}+\cdots+2^{i_k}} = a^{2^{i_1}}a^{2^{i_2}}\cdots a^{2^{i_k}}.$$
So we only need to compute a^j, where $j \le i_1$ is a power of 2. This can be done by computing a^2, $a^4 = a^2a^2$, $a^8 = a^4a^4$, etc., until all the desired powers of a have been computed. Moreover, since it takes k multiplications to reach 2^k, we see that $\Theta(\log n)$ multiplications are needed.

10. Use an algorithm similar to binary search. If mid is the midpoint index, what can you say if $a[\text{mid}] > \text{mid}$? Why? What can you say if $a[\text{mid}] < \text{mid}$? Why?

11. First assume that the array has no zeros (if the array does have zeros, then divide it into subarrays that have not zero, and apply your algorithm to each subarray). In this case, the largest (absolute-value) product is simply the product of the entire array. How to trim this product in order to get the largest positive product?

12. For a given problem input, first test to see if the input represents a special case that can be easily solved. For example, if the problem is to sort an array, then first check if the array is already sorted. This takes $O(n)$ steps to check, which is typically better than the worst-case time for most sorting algorithms.