CS201: Data Structures and Discrete Math I

Algorithm (Run Time) Analysis

Motivation

- Purpose: Understanding the resource requirements of an algorithm
  - Time
  - Memory
- Running time analysis estimates the time required of an algorithm as a function of the input size.
- Usages:
  - Estimate growth rate as input grows.
  - Guide to choose between alternative algorithms.

An example

-
  ```
  int sum(int set[], int n) {
          int temsum, i;
          tempsum = 1;        /* step/execution 1 */
          for (i=0; i<n; i++)            /* step/execution n+1 */
              tempsum +=set[i]; /* step/execution n */
          return tempsum;                /* step/execution 1 */
  }
  ```
- Input size: n (number of array elements)
- Total number of steps: $2n + 3$

Analysis and measurements

- Performance measurement (execution time): **machine dependent**.
- Performance analysis: **machine independent**.
- How do we analyze a program independent of a machine?
  - Counting the number steps.

Model of Computation

- Model of computation is an ordinary (sequential) computer
- Assumption: basic operations (steps) take 1 time unit.
- What are basic operations?
  - Arithmetic operations, comparisons, assignments, etc.
  - Library routines such as *sort* should not be considered basic.
  - Use common sense

Big-Oh Notation

- A standard for expressing upper bounds
- **Definition**: $T(n) = O(f(n))$ if there exist constant c and n_0 such that $T(n) \leq cf(n)$ for all $n \geq n_0$
  - We say: *T(n)* is big-O of *f(n), or*
    The time complexity of *T(n)* is *f(n).*
- Intuitively, an algorithm A is $O(f(n))$ means that, if the input is of size n, the algorithm will stop after $f(n)$ time.
- The running time of *sum is O(n), i.e.,* ignore constant 2 and value 3 ($T(n)= 2n + 3$).
  - because $T(n) \leq 3n$ for $n \geq 10$ ($c = 3$, and $n_0 = 10$)

Example 1

- Definition does not require upper bound to be tight, though we would prefer as tight as possible
- What is Big-Oh of $T(n) = 3n+3$
  - Let $f(n) = n$, $c = 6$ and $n_0 = 1$;

    $T(n) = O(f(n)) = O(n)$ because $3n+3 \leq 6f(n)$ if $n \geq 1$
  - Let $f(n) = n$, $c = 4$ and $n_0 = 3$;

    $T(n) = O(f(n)) = O(n)$ because $3n+3 \leq 4f(n)$ if $n \geq 3$
  - Let $f(n) = n^2$, $c = 1$ and $n_0 = 5$;

    $T(n) = O(f(n)) = O(n^2)$ because $3n+3 \leq (f(n))^2$ if $n \geq 5$
- We certainly prefer $O(n)$.

Example 2

- What is Big-Oh for $T(n) = n^2 + 5n - 3$?
  - Let $f(n) = n^2$, $c = 2$ and $n_0 = 6$.
    Then $T(n) = O(f(n)) = O(n^2)$ because

    $$T(n) \leq 2\, f(n) \text{ if } n \geq n_0.$$

    i.e., $n^2 + 5n - 3 \leq 2n^2$ if $n \geq 6$
  - Can we find $T(n) = O(n)$? No, we cannot find c and n_0 such that $T(n) \leq c\, n$ for $n \geq n_0$. Why?

    $$\lim_{n \to \infty} T(n)/n \to \infty$$

Rules for Big-Oh

- If $T(n) = O(c\ f(n))$ for a constant c, then

  $T(n) = O(f(n))$
- If $T_1(n) = O(f(n))$ and $T_2(n)=O(g(n))$ then

  $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$
- If $T_1(n) = O(f(n))$ and $T_2(n)=O(g(n))$ then

  $T_1(n) * T_2(n) = O(f(n) * g(n)))$
- If $T(n) = a_m n^k + a_{m-1} n^{k-1} + \dots + a_1 n + a_0$ then

  $T(n) = O(n^k)$
- Thus
  - Lower-order terms can be ignored.
  - Constants can be thrown away.

More about Big-Oh notation

- Asymptotic: Big-Oh is meaningful only when n is sufficiently large

  $n \geq n_0$ means that we only care about large size problems.
- Growth rate: A program with O(f(n)) is said to have growth rate of f(n). It shows how fast the running time grows when n increases.

Typical bounds (Big-Oh functions)

- Typical bounds in increasing order of growth rate

Function	Name
$O(1)$,	Constant
$O(\log n)$,	Logarithmic
$O(n)$,	Linear
$O(n\log n)$,	Log linear
$O(n^2)$,	Quadratic
$O(n^3)$,	Cubic
$O(2^n)$	Exponential

Growth rates illustrated

	n=1	n=2	n=4	n=8	n=16	n=32
$O(1)$	1	1	1	1	1	1
$O(logn)$	0	1	2	3	4	5
$O(n)$	1	2	4	8	16	32
$O(nlogn)$	0	2	8	24	64	160
$O(n^2)$	1	4	16	64	256	1024
$O(n^3)$,	1	8	64	512	4096	32768
$O(2^n)$	2	4	16	235	65536	4294967296

Exponential growth

- Say that you have a problem that, for an input consisting of n items, can be solved by going through 2^n cases
- You use Deep Blue, that analyses *200* million cases per second
  - Input with *15* items, *163* microseconds
  - Input with *30* items, *5.36* seconds
  - Input with *50* items, more than two months
  - Input with *80* items, *191* million years

How do we use Big-Oh?

- Programs can be evaluated by comparing their Big-Oh functions with the constants of proportionality neglected. For example,
  - $T_1(n) = 10000\ n$ and $T_2(n) = 9\ n$. The time complexity of $T_1(n)$ is equal to the time complexity of $T_2(n)$.
- The common Big-Oh functions provide a "yardstick" for classifying different algorithms.
- Algorithms of the same Big-Oh can be considered as equally good.
- A program with *O(log n)* is better than one with *O(n)*.

Nested loops

- Running time of a loop equals running time of the code within the loop times the number of iterations.
- Nested Loops: analyze inside out

```
1  for (i=0; i <n; i++)
2       for (j = 0; j< n; j++)
3                k++
```

- Running time of lines 2-3: $O(n)$
- Running time of lines 1-3: $O(n^2)$

Consecutive statements

- For a sequence S1, S2, .., Sk of statements, running time is maximum of running times of individual statements

```
for (i=0; i<n; i++)
        x[i] = 0;
for (i=0; i<n; i++)
        for (j=0; j<n; j++)
                k[i] += i+j;
```

- Running time is: $O(n^2)$

Conditional statements

- The running time of

  If (cond) S1

  else S2

  is running time of *cond* plus the max of running times of S1 and S2.

More nested loops

```
1    int k = 0;
2    for (i=0; i<n; i++)
3        for (j=i; j<n; j++)
4            k++
```

- Running time of lines 3-4: n-i
- Running time of lines 1-4:

$$\sum_{i=0}^{n-1}(n-i)=n(n+1)/2=O(n^2)$$

More nested loops

```
int k = 0;
for (i=1; i<n; i*= 2)
    for (j=1; j<n; j++)
        k++
```

- Running time of inner loop: O(n)
- What about the outer loop?
- In *m*-th iteration, value of i is 2^{m-1}
- Suppose $2^{q-1} < n \leq 2^{q}$, then outer loop is executed q times.
- Running time is O(n log n). Why?

A more intricate example

```
int k = 0;
for (i=1; i<n; i*= 2)
    for (j=1; j<i; j++)
        k++
```

- Running time of inner loop: O(i)
- Suppose $2^{q-1} < n \le 2^q$, then the total running time:

  $1 + 2 + 4 + \ldots + 2^{q-1} = 2^q - 1$
- Running time is O(n).

Lower Bounds

- To give better performance estimates, we may also want to give lower bounds on growth rates
- Definition (omega): $T(n) = \Omega(f(n))$

  if there exist some constants c and n_0 such that $T(n) \geq cf(n)$ for all $n \geq n_0$

“Exact” bounds

- Definition (Theta): $T(n) = \Theta(f(n))$ if and only if $T(n) = O(f(n))$ and $T(n) =$ $(f(n))$.
- An algorithm is $\Theta(f(n))$ means that $f(n)$ is a tight bound (as good as possible) on its running time.
  - On all inputs of size n, time is $\leq f(n)$
  - On all inputs of size n, time is $\geq f(n)$

```
int k = 0;
for (i=1; i<n; i*=2)
        for (j=1;j<n; j++)
                k++
```

This program is $O(n^2)$ but not (n^2); it is $\Theta(n \log n)$

Computing Fibonacci numbers

- We write the following program: a recursive program

```
long int fib(n) {
 if (n <= 1)
     return 1;
 else return fib(n-1) + fib(n-2)
```

- Try fib(100), and it takes forever.
- Let us analyze the running time.

fib(n) runs in exponential time

- Let T denote the running time.

  $T(0) = T(1) = c$

  $T(n) = T(n-1) + T(n-2) + 2$

  where 2 accounts for line 2 plus the addition at line 3.
- It can be shown that the running time is $((3/2)^n)$.
- So the running time grows exponentially.

Efficient Fibnacci numbers

- Avoid recomputation
- Solution with linear running time

```
int fib(int n)
{
   int fibn=0, fibn1=0, fibn2=1;

   if (n < 2)
       return n
    else
      {
         for( int i = 2; i <= n; i++ ) {
             fibn = fibn1 + fibn2;
             fibn1 = fibn2;
             fibn2 = fibn;
         }
      return fibn;
      }
}
```

What happens in practice

- We ignore many important factors that will determine the actual running time.
  - Speed of processor
  - Constants are ignored
  - Fine-tuning by programmers
  - Different basic operations take different times,
  - Load, I/O, available memory
- In spite of above, $O(n)$ algorithms will outperform $O(n^2)$ algorithm for “large enough” input
- $O(2^n)$ algorithm will never work on large inputs.

Maximum subsequence sum problem

- Input: array X of n integers (can be negative)
  - E.g.: 2 6 -3 -7 5 -2 4 -12 9 -4
- Output: find a subsequence with maximum sum, i.e., find $0 \le i \le j < n$ to maximize

$$\sum_{k=i}^{j} X[k]$$

- Assumption: if all are negative, then output is 0
- The problem is interesting because different algorithms have very different running times.

First solution

- For every pair (i, j) $(0 \le i \le j < n)$, compute sum $$\sum_{k=i}^{j} X[k]$$
- It does not produce the actual subsequence.

```
MSS1 (int X[ ], int n) {
        int current = 0, i, j, k, result = 0;
        for (i = 0; i<n; i++)
          for (j=i; j<n; j++) {
              current = 0;
              for (k = i; k<=j; k++)
                     current +=X[k];
              if (current > result)
                 result = current;
              }
       return result; }
```

Analysis of MSS1

- Just look at the three nested loops: O(n^3). Can we get a better bound?
- Number of iteration of innermost loop (line 7) is j – i + 1
- Running time of lines 4-10:

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

- The total running time:

$$\sum_{i=0}^{n-1} \frac{(n-i)(n-i+1)}{2}=\frac{n^3+3\mathrm{n}^2+2\mathrm{n}}{6}$$

- Running time is Θ (n^3)

A Quadratic Solution

- Observation: Sum of X[i..(j+1)] can be computed by adding X[j+1] to sum of X[i..j]
- MSS2 has $\Theta(n^2)$ running time

```
MSS1 (int X[ ], int n) {
        int current = 0, result = 0, i, j, k;
        for (i = 0; i<n; i++) {
            current = 0;
            for (j=i; j<n; j++) {
                current +=X[j];
                if (current > result)
                    result = current;
            }   }
       return result; }
```

A recursive solution

- Divide the problem in two parts: find maximum subsequences of left and right halves, and take the maximum of the two.
- This, of course, is not sufficient. Why?
- We need to consider the case when the desired subsequence spans both halves.

The recursive program

```
MSS3 (int X [ ], int n) {
    return RMSS (X, 0, n-1) }

RMSS (int X [ ], int Left, int Right) {
    if (Left == Right) return (max(X[Left], 0));
    int Center = (Left + Right)/2;
    int maxLeftSum = RMSS(X, Left, Center);
    int maxRightSum = RMSS(X, Center +1, Right);
    int current = result = X[Center];
    for (int i = Center -1; i >= Left; i--) {
            current += X[i];
            result = max(result, current); }
    current = result = result + X[Center +1];
    for (i = Center + 2; i < Right; i++) {
            current +=X[i];
            result = max(result, current); }
    return (max (maxLeftSum, maxRightSum, result)); }
```

Analysis of MSS-3

- Let $T(n)$ be running time of RMSS.
- Base case: $T(1) = O(1)$
- Recursive case:
  - Two recursive calls of size $n/2$
  - Plus $O(n)$ work for the rest of the code
- This gives

  $T(1) = O(1)$, $T(n) = 2T(n/2) + O(n)$
- It turns out that $n = 2^k$, $T(n) = nk+n$ satisfy the equation.
- Running time $T(n) = n\log n + n = O(n \log n)$

An even better solution

- Let us call position j a breakpoint if the sums X[i..j] are negative for all $0 \le i \le j$.
- Example, 2 6 -3 -7 5 -2 4 -12 9 -4
- Property 1: Max subsequence won't include a breakpoint.
- If j is a breakpoint, then solution is max of the solutions of the two halves X[0..j] and X[j+1..n-1]
- Property 2: If j is the least position such that the sum X[0..j] is negative, then j is a breakpoint.

The solution

```
MSS4 (int X [ ], int n) {
    int current = 0, result = 0;
    for (int j=0; j<n;j++) {
        current += X[j];
        result = max (result, current);
        if (current < 0)
           current = 0;
    }
    return result;
}
```

- A single loop: running time is O(*n*).

Linear search

- Input: array A contains n integers, already sorted in increasing order, and an integer x.
- Output: Is x an element of the array?
- Linear search: scan the array left to right.

```
linear_search(int A[], int x, int n)
        for (i=0; i<n; i++) {
                if (A[i] == x) return i;
                if (A[i] > x) return Not_found
        }
        return Not_found;
}
```

- Running time (worst case): $O(n)$
- If constant time is needed to merely reduce the problem by a constant amount, then the algorithm is *O*(n).

Binary search (the same problem)

- Binary search: locate the midpoint, decide whether x belongs to left half or right half, and repeat in the appropriate half.

```
Binary_search(int A [], int x, int n)
     int low =0, high=n-1, mid;
   while (low <= high ) {
               mid = (low + high ) / 2
               if (A[mid]<x)  low = mid+ 1;
               else if (A[mid]> x)  high = mid -1;
               else return mid; }
    return Not_Found; }
```

- Total time: O(log n)
- An algorithm is O(log n) if it takes constant time to cut the problem size by a fraction (usually ½).

Euclid’s algorithm

- Compute greatest common divisor

```
GCD(int m, int n)
{
    int rem;
    while ( n != 0) {
        rem = m % n;
        m = n;
        n = rem;   }
    return m;
}
```

Sample execution:

m= 1203	n=522	rem = 159
m= 522	n=159	rem = 45
m= 159	n=45	rem = 24
m= 45	n=24	rem = 21
m= 24	n=21	rem = 3
m= 21	n=3	rem = 0
m= 3	n=0	

Analysis of Euclid's algorithm

- Correctness: if m > n > 0 then

  GCD(m, n) = GCD(n, m mod n)

- Theorem: If m>n then m mod n < m/2
- It follows that the remainder decrease by at least a factor of 2 every two iterations
- Number of iterations: 2 log n
- Running time: O(log n)

Summary: lower vs. upper bounds

- This section gives some ideas on how to analyze the complexity of programs.
- We have focused on worst case analysis.
- Upper bound O(f(n)) means that for sufficiently large inputs, running time T(n) is bounded by a multiple of f(n).
- Lower bound Ω(f(n)) means that for sufficiently large n, there is at least one input of size n such that running time is at least a fraction of f(n)
- We also touch the “exact” bound Θ(f(n)).

Summary: algorithms vs. Problems

- Running time analysis establishes bounds for individual algorithms.
- Upper bound O(f(n)) for *a problem*: there is some O(f(n)) algorithms to solve the problem.
- Lower bound Ω(f(n)) for *a problem*: every algorithm to solve the problem is Ω(f(n)).
- They different from the lower and upper bound of an algorithm.