Algorithms

Lecture 1

Timothy Kim
GWU CSCI 6212

Algorithm Definition

- Well-defined computational procedure that solves a problem
    - Takes input
    - Produces output
- Single problem can have multiple algorithm that solves it

Characteristics

- Definiteness
- Effectiveness
  - Finite Number of instructions
  - Terminates (finite runtime steps taken)
  - Produces correct output
  - Requires rigorousness not ingenuity

Sample Problem

- Given *n* numbers, find the maximum and the minimum values

MaxMin Algorithm 1

```
MaxMin(S_n):
  max = S[1]
  for i = 2 to n:
    if (S[i] > max)
      max = S[i]
  S' = S - max
  min = S'[1]
  for i = 2 to n-1
    if (S'[i] < min)
      min = S'[i]
  return max, min
```

MaxMin Algorithm 2

```
MaxMin(Sn)
  if |S| = 2
    return Max(S[1], S[2]), Min(S[1], S[2])
  else
    S'  = S[1] ... S[n/2]
    S'' = S[n/2 + 1] ... S[n]
    max1, min1 = MaxMin(S')
    max2, min2 = MaxMin(S'')
    return Max(max1, max2), Min(min1, min2)
```

Which one is faster?

- How do we compare?
- What metrics do we use?

Analyzing an Algorithm

- Assumptions:
  - Random access memory model
  - Math operations take constant time
  - Read/write operations take constant time
- We want to compute $T(n)$ or running time: number of operations as a function of input size.

MaxMin Algorithm 1

```
MaxMin(Sn):
  max = S[1]                     1
  for i = 2 to n:                3 instructions n times
    if (S[i] > max)
      max = S[i]
  S' = S - max                   1
  min = S'[1]                    1
  for i = 2 to n-1               3 instructions n times
    if (S'[i] < min)
      min = S'[i]
  return max, min                1
```

Computational Steps

- MaxMin Algorithm 1
  - $T(n) = 6n + 4$
- MaxMin Algorithm 2
  - recursion ???

MaxMin Algorithm 2

```
MaxMin(Sn)                                            T(n)
  if |S| = 2
    return Max(S[1], S[2]), Min(S[1], S[2])           2
  else
    max1, min1 = MaxMin(S[1] ... S[n/2])              T(n/2)
    max2, min2 = MaxMin(S[n/2 + 1] ... S[n])          T(n/2)
    return Max(max1, max2), Min(min1, min2)           2
```

Computational Steps

MaxMin Algorithm 1

$$T(n) = 6n + 4$$

MaxMin Algorithm 2

$$T(n) = \begin{cases} 3 & \text{if } n \le 2 \\ 2\,T(n/2) + 2 & \text{if } n > 2 \end{cases}$$

(recurrence)

Solving the Recurrence by Subsitution

```
T(n) = 2 T(n/2) + 2
     = 2 [ 2 T(n/4) + 2 ] + 2     (substitution)
     = 4 T(n/4) + 4 + 2           (expand)
     = 4 [ 2 T(n/8) + 2 ] + 4 + 2 (another substitution)
     = 8 T(n/8) + 8 + 4 + 2
     ...
     = 2^k T(2) + 2^k + 2^k-1 + ... + 4 + 2
                                  (after k substitution
                                   where k = log₂n)
     = 2^k * 3 + 2^(k+1) - 2
     = 3*2^k + 2 (2^k - 1)        (note: 2^k = n)
     = 3n + 2(n-1) = 3n + 2n - 2
     = 5n - 2
```

Computational Steps Final

```
MaxMin Algorithm 1

 T(n) = 6n + 4


MaxMin Algorithm 2

 T(n) = 3               if n ≤ 2
      = 2 T(n/2) + 2  if n > 2  (recurrence)
      or
      = 5n - 2
```

Analysis Method 1

- Average Case Analysis
  - Average running time of every possible input of length *n*
- Worst Case Analysis
  - Running time **bound** that holds for ALL possible input length of *n*

Analysis Method 2

- Ignore Constant Factors
  - Easier
  - Architecture and implementation dependent
  - We don't gain much by including them

Analysis Method 3

- Asymptotic Analysis
  - Assume *n* is very large
  - Why?

So how do you measure FAST?

- Fast = Worst Case running time grows slowly with input size
- Usually we want O(n)

O of what?

Formal definition

f(n) = O(g(n)) if and only if

$$\exists\ c > 0,\ n_o > 1 \mid f(n) \leq cg(n)\ \forall\ n \geq n_o$$

In math english

f(n) is considered to be O(g(n)) if and only if
there exists a constant c > 0 and $n_o > 1$
such that for all $n > n_o$, $f(n) \leq cg(n)$ holds

In plain english

- If we say an algorithm's runtime $T(n) = O(g(x))$,
  then eventually ($n \geq n_o$), $T(n)$ is bounded above by a constant multiple of $g(x)$

- More practically, if $f(x) = O(g(x))$ and $h(x) = O(g(x))$, then $f(x)$ and $h(x)$ are both bounded by $g(x)$ thus in the same family of algorithms that share similar runtime property

Theorem

Let $A(n) = a_m n^m + a_{m-1} n^{m-1} + ... + a_1 n + a_0$

then $A(n) = O(n^m)$

Proof:

$$
\begin{aligned}
A(n) \le |A(n)| &\le |a_m| n^m + |a_{m-1}| n^{m-1} + ... + |a_1| n + |a_0| \\
&\le |a_m| n^m + |a_{m-1}| n^m + ... + |a_1| n^m + |a_0| n^m \\
&= (|a_m| + |a_{m-1}| + ... + |a_1| + |a_0|) n^m \\
&= c \cdot n^m \text{ for any value of } n \ge 1
\end{aligned}
$$

$$\therefore A(n) = O(n^m)$$

Practical Application:

if $T(n) = n^4 + 3n - 4$, then $T(n) = O(n^4)$

Corollary

Claim:

$\forall\ k \geq 1,\ n^k \neq O(n^{k-1})$

Proof by contradiction:

Assume the opposite, suppose $n^k = O(n^{k-1})$

Then $\exists\ c > 0,\ n_o > 1 \mid n^k \leq cn^{k-1}\ \forall\ n \geq n$

Take $n^k \leq cn^{k-1}$ and divide each side by n^{k-1}

$n \leq c$

n cannot be bounded by some constant c.

In other words n can equal to $c + 1$,

which is a contradiction.

Thus proving our initial claim.

More definition

$f(\mathrm{n}) = \Omega(g(n))$ if and only if

$$\exists\ c > 0,\ n_o > 1 \mid f(n) \geq cg(n)\ \forall\ n \geq n_o$$

$f(\mathrm{n}) = \Theta(g(n))$ if and only if

$$f(\mathrm{n}) = O(g(n)) \text{ and } f(\mathrm{n}) = \Omega(g(n))$$

Properties

Let $f(n) = O(s(n))$ and $g(n) = O(t(n))$

Then

$$f(n) + g(n) = O(s(n) + t(n))$$
$$f(n) \cdot g(n) = O(s(n) \cdot t(n))$$

But not necessarily

$$f(n) - g(n) = O(s(n) - t(n))$$
$$f(n) / g(n) = O(s(n) / t(n))$$

Sample Problem

Prove that $2^{n+3} = O(2^n)$

Proof:

We need to find c and n_o such that

$$2^{n+3} \le c \cdot 2^n \quad \text{for } n \ge n_o$$

$$2^n \cdot 2^3 \le c \cdot 2^n$$

$$2^3 \le c \quad \text{for all } n \ge 1 \text{ thus } n_o = 1$$