L12: Analysis of Algorithms

- Reading: Rosen 3.1, 3.2, 3.3

Revisiting the Selection Sort Algorithm

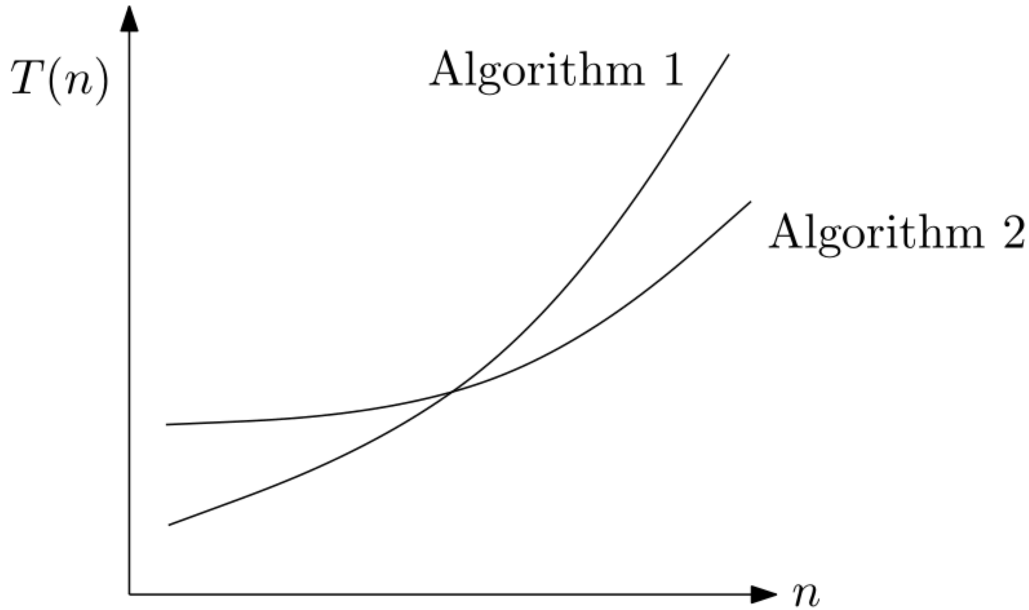
```
for i = 1 to n - 1
    for j = i + 1 to n
        if ( A[i] > A[j] )
            swap A[i] and A[j]
        endif
    endfor
endfor
```

An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

How to Measure the Running Time?

- The real running time when executed on a computer?
- The number of times a particular line is executed (as a function of input size n)?
  - We counted line (3). Answer: $n(n-1)/2$
- The total number of lines executed?
- The total number of machine instructions, but…
  - Ignore lower order terms
  - Ignore constant coefficients
    - A constant is any quantity that doesn't depend on n.
- E.g., the running time of selection sort is $\Theta(n^2)$

The Growth of Functions



- Which algorithm is better for large n?
  - For Algorithm 1, $T_1(n) = 3n^3 + 6n^2 - 4n + 17 = \Theta(n^3)$
  - For Algorithm 2, $T_2(n) = 7n^2 - 8n + 20 = \Theta(n^2)$
  - Clearly, Algorithm 2 is better

Big-Theta

- **Definition**: Let f and g be functions from the set of positive real numbers to the set of positive real numbers. We say that $f(x)$ is $\Theta(g)$ if there are constants C_1, C_2, and k such that
$$C_1 g(x) \leq f(x) \leq C_2 g(x)$$
whenever $x > k$.
- This is read as "$f(x)$ is big-Theta of $g(x)$" or "$f(x)$ is asymptotically the same as $g(x)$."
- Usually written as $f(n) = \Theta\big(g(n)\big)$, although the more mathematically correct way should be $f(n) \in \Theta\big(f(n)\big)$.
- The constants C_1, C_2 and k are called *witnesses* to the relationship. Only one pair of witnesses is needed.

Using Definition to Derive Big-Theta

$$T_1(n) = 3n^3 + 6n^2 - 4n + 17 = \Theta(n^3)$$

- Choose $C_1 = 2, C_2 = 4$
- Want a k such that, when $n > k$

$$2n^3 \le 3n^3 + 6n^2 - 4n + 17 \le 4n^3$$
$$-n^3 \le 6n^2 - 4n + 17 \le n^3$$

- It's clear that such a k must exist, there is no need to actually find it.

Comparison of Algorithms

- n is big (big data!), so we are interested in
$$\lim_{n\to\infty} \frac{T_1(n)}{T_2(n)}$$
- Three cases:
  - $\lim_{n\to\infty} \frac{T_1(n)}{T_2(n)} = 0$: Algorithm 1 is better
  - $\lim_{n\to\infty} \frac{T_1(n)}{T_2(n)} = \infty$: Algorithm 2 is better
  - $\lim_{n\to\infty} \frac{T_1(n)}{T_2(n)} = C$ for some constant $0 < C < \infty$, or $\frac{T_1(n)}{T_2(n)}$ oscillates: Θ-notation cannot tell, need more careful analysis.
- If $T_1(n) = \Theta\big(g_1(n)\big), T_2(n) = \Theta\big(g_2(n)\big)$, it's sufficient to consider $\lim_{n\to\infty} \frac{g_1(n)}{g_2(n)}$

Examples

- $\log_{10} n = \frac{\log_2 n}{\log_2 10} = \Theta(\log_2 n) = \Theta(\log n)$
- $9999^{9999^{9999}} = \Theta(1)$
- 2^{10n} is not $\Theta(2^n)$, 3^n is not $\Theta(2^n)$
- $\sum_{i=1}^{n} i^2 \leq n^2 \cdot n \leq n^3$
  $\sum_{i=1}^{n} i^2 \geq \left(\frac{n}{2}\right)^2 \cdot \left(\frac{n}{2}\right) \geq \frac{1}{8} n^3$
  So, $\sum_{i=1}^{n} i^2 = \Theta(n^3)$
- $\sum_{i=0}^{n} c^i = \frac{c^n - 1}{c - 1} = \begin{cases} \Theta(c^n), c > 1 \\ \Theta(n), c = 1 \\ \Theta(1), c < 1 \end{cases}$ (geometric series)

Solving Geometric Series

- Geometric series

$$S(n) = 1 + c + c^2 + c^3 + \cdots + c^n$$

- Solving geometric series

$$S(n+1) = S(n) \cdot c + 1$$
$$S(n+1) = S(n) + c^{n+1}$$
$$S(n) \cdot c + 1 = S(n) + c^{n+1}$$
$$(c-1)S(n) = c^{n+1} - 1$$

- If $c \neq 1$,

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

Examples

- $\log(n!) = \log(n) + \log(n-1) + \cdots + \log 1 \leq n \log n$
  $\log(n!) \geq \log(n) + \log(n-1) + \cdots + \log\left(\frac{n}{2}\right) \geq \frac{n}{2} \log\left(\frac{n}{2}\right)$
  So, $\log(n!) = \Theta(n \log n)$.
- $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$ (harmonic series, derivation on board)
- c_1^n dominates n^{c_2}, which dominates $\log^{c_3} n$,
  for $c_1 > 1, c_2 > 0, c_3 > 0$
- $n^{0.1} + \log^{10} n = \Theta(n^{0.1})$
- $1.1^n + n^{100} = \Theta(1.1^n)$

Limitation of Big-Theta

- Some functions cannot be described by Θ
  - Example:
    the number of 1's in the binary representation of n
  - Oscillates between 1 and $\log n$
- Some functions are hard to describe by Θ
  - Example: $n!$
  - It is known that $n! = \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right)$ (Stirling's formula)
    but it's very difficult to derive
- When used for analysis of algorithms (later)

Big-Oh

- **Definition**: Let f and g be functions from the set of positive real numbers to the set of positive real numbers. We say that $f(x)$ is $\mathrm{O}(g)$ if there are constants C_2, and k such that

$$f(x) \leq C_2 g(x)$$

whenever $x > k$.
- This is read as "$f(x)$ is big-Oh of $g(x)$" or "$f(x)$ is asymptotically dominated by $g(x)$."
- Usually written as $f(n) = O\big(g(n)\big)$, although the more mathematically correct way should be $f(n) \in O\big(f(n)\big)$.
- The constants C_2 and k are called *witnesses* to the relationship. Only one pair of witnesses is needed.

Big-Omega

- **Definition**: Let f and g be functions from the set of positive real numbers to the set of positive real numbers. We say that $f(x)$ is $\Omega(g)$ if there are constants C_1, and k such that
$$C_1 g(x) \leq f(x)$$
whenever $x > k$.
- This is read as "$f(x)$ is big-Omega of $g(x)$" or "$f(x)$ asymptotically dominates $g(x)$."
- Usually written as $f(n) = \Omega\big(g(n)\big)$, although the more mathematically correct way should be $f(n) \in \Omega\big(f(n)\big)$.
- The constants C_1 and k are called *witnesses* to the relationship. Only one pair of witnesses is needed.
- $f(x) = \Theta\big(g(x)\big)$ iff $f(x) = O\big(g(x)\big)$ and $f(x) = \Omega\big(g(x)\big)$

Examples:

- $f(n) = 32n^2 + 17n - 32$.
  - $f(n)$ is $O(n^2), O(n^3), \Omega(n^2), \Omega(n)$, and $\Theta(n^2)$.
  - $f(n)$ is not $O(n), \Omega(n^3), \Theta(n)$, or $\Theta(n^3)$.
- The number of 1's in the binary representation of n is
  - $O(\log n)$
  - $\Omega(1)$
- $n!$
  - $n! \le n^n = O(n^n)$
  - $n! \ge \left(\frac{n}{2}\right)^{\frac{n}{2}} = \Omega\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right)$

Insertion Sort

<u>**Insertion-Sort**(A)**:**</u>

for $j \leftarrow 2$ **to** n **do**
    $key \leftarrow A[j]$
    $i \leftarrow j - 1$
    while $i \geq 1$ **and** $A[i] > key$ **do**
        $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    endwhile
    $A[i+1] \leftarrow key$
endfor

sorted	key	unsorted

Insertion Sort: Example

- 1st iteration:
  - (4 1 8 2 5) → (4 4 8 2 5) → (1 4 8 2 5)
  - key = 1
- 2nd iteration:
  - (1 4 8 2 5)
  - key = 8
- 3rd iteration:
  - (1 4 8 2 5) → (1 4 8 8 5) → (1 4 4 8 5) → (1 2 4 8 5)
  - key = 2
- 4th iteration:
  - (1 2 4 8 5) → (1 2 4 8 8) → (1 2 4 5 8)
  - key = 5

Analysis of Algorithms

- Question: What’s the running time of insertion sort if the input array is already sorted?
- Answer: $\Theta(n)$.
- Question: What’s the running time of insertion sort if the input array is inversely sorted?
- Answer: $\Theta(n^2)$.
- Observation: The running time of an algorithm doesn’t only depend on n, it also depends on the actual input!
- Question: Which input should we use for analyzing an algorithm?
  - Best-case? Average-case? Worst-case?

Worst-case Analysis

- The default of algorithm analysis
- Especially easy when using big-Oh
  - You don’t really have to find the worst-case input!
  - Can easily conclude that insertion sort runs in time $O(n^2)$.
- How to show an algorithm has worst-case running time $\Theta(n^2)$?
  - Show that it’s $O(n^2)$
  - Show that it’s $\Omega(n^2)$
    - Find an input such that the running time is $\Omega(n^2)$

Example: Linear Search

- Problem: Given an array A of unordered elements and x, find x or report that x doesn't exist in A
- Algorithm:

procedure *linear search*(x:integer,
$a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
    $i := i + 1$
if $i \leq n$ **then** *location* $:= i$
else *location* $:= 0$
return *location*

Analysis of Linear Search

- Running time is $O(n)$
  - Obvious, since the loop has at most n iterations.
- (Worst-case) running time is $\Omega(n)$
  - When x doesn't exist in A, the loop has exactly n iterations.
- Running time is $\Theta(n)$

Example: Binary Search

- Problem: Given an array A of ordered elements and x, find x, or report that x doesn't exist in A
- Algorithm:

procedure binary search(x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1$ {i is the left endpoint of interval}
$j := n$ {j is right endpoint of interval}
while $i < j$
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ then $i := m + 1$
    else $j := m$
if $x = a_i$ **then** *location* $:= i$
else *location* $:= 0$
return *location*

Analysis of Binary Search

- Assumption: $n = 2^k$
- Running time is $O(k) = O(\log n)$
  - The length of the range $j - i + 1$ decrease by half in each iteration
  - The algorithm terminates when $i \geq j$, i.e.,
$$j - i + 1 \leq 1$$
- (Worst-case) running time is $\Omega(\log n)$
  - When x doesn't exist in A, the loop has exactly $\log n$ iterations.
- Running time is $\Theta(\log n)$
- What if n is not in the form of 2^k?
  - Find k such that $2^k < n < 2^{k+1}$. The running time must be between $\Theta(k)$ and $\Theta(k+1)$, which is $\Theta(k)$.