CSCE 750 — Analysis of algorithms

Algorithms ⟷ Data Structs

Asymptotic Analysis

Def: $f : \mathbb{N} \to \mathbb{R}$ is *asymptotically positive* if $\exists n_0 \in \mathbb{N}\ \forall n \geq n_0,\ f(n) > 0$.

($\mathbb{N}$ = nat nums $\{0, 1, 2, \ldots\}$; $\mathbb{R}$ = reals)

Def: Let $f, g : \mathbb{N} \to \mathbb{R}$ be asymptotically positive. We say that

- $f(n) = O(g(n))$ if $\exists c > 0$, $\exists n_0 \in \mathbb{N}\ \forall n \geq n_0$,
  $$f(n) \leq c \cdot g(n)$$
  $[f \in O(g)]$

- $f(n) = \Omega(g(n))$ if $\exists c > 0$, $\exists n_0 \in \mathbb{N}$, $\forall n \geq n_0$
  $$f(n) \geq c \cdot g(n)$$

- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

"*f & g are asymptotically similar*"

$f(n) = \Omega(g(n))$ iff $g(n) = O(f(n))$

"$f(n) = O(g(n))$" binary relation on asymp. pos. functions.

- reflexive: $f(n) = O(f(n))$
- transitive: $f(n) = O(g(n))$ & $g(n) = O(h(n))$ $\Rightarrow f(n) = O(h(n))$

$\therefore$ "$f(n) = \Theta(g(n))$" is an equiv relation (reflexive, symmetric, transitive)

---

strict versions of O & Ω:

$f(n) = \omega(g(n))$ means
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

$\forall c > 0, \exists n_0, \forall n \geq n_0, f(n) > c \cdot g(n)$

$f(n) = o(g(n))$ means
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

[equiv: $g(n) = \omega(f(n))$]

Ignore constants — don't care about variations in implementation (software or hardware)

Ignore small inputs (small n) not sufficient to determine major differences among alternative algorithms.

```
sum = 0        } O(1)
i = 1
while i <= n:              O(1)
    for j in range(1, i):
        sum = sum + 1      ] O(1)   } Θ(i)
    i = i * 2      ] O(1)
```

(lg n times)

versus

```
sum = 0
i = 1
while i <= n:
    for j in range(i, n):
        sum = sum + 1
    i = i * 2
```

(lg n times) $\Theta(n - i) = O(n)$

$\lg = \log_2$

$\ln = \log_e$

Primitive ops:

- $+, *, -, /, \%$ on "reasonably small" integers
- comparisons of reas. small integers
- assignments " " " "
- array indexing

Assume all primitive ops take $O(1)$ (constant time) each.

[*reasonably small* means requiring $O(\log n)$ bits to represent, where n represents the size of the input.]

First analysis: both take time $O(n \lg n)$. tight for the 2nd code but not the first.

Redo the 1st code analysis:

$$\text{Time} = \Theta\left(\sum_{k=0}^{\lg n} \underbrace{2^k}_{i}\right)$$

$$\sum_{k=0}^{\ell} 2^k = 2^{\ell+1} - 1 = 2^{1 + \lg n} - 1$$

$[\ell = \lg n]$:

$$= 2 \cdot 2^{\lg n} - 1$$
$$= 2n - 1 = \Theta(n)$$

---

multiplying two n-bit numbers (natural numbers)

Two approaches, both divide-and-conquer

Naive approach: $\Theta(n^2)$ time

A, B are the numbers

h = high
l = low

$$A = A_h \cdot 2^{n/2} + A_l$$
$$B = B_h \cdot 2^{n/2} + B_l$$

$$AB = (A_h 2^{n/2} + A_l)(B_h \cdot 2^{n/2} + B_l)$$
$$= A_h B_h 2^n + (A_h B_l + A_l B_h) 2^{n/2} + A_l B_l$$

recursive call on 4 pairs of numbers, each of size $n/2$

$\times 2^n$ $\times 2^{n/2}$ one or the other assume $O(n)$ each (probably much faster than this)

+3 additions of n-bit number take total time $\Theta(n)$ outside of the recursive calls.

Let $T(n)$ be the time this algo takes of 2 n-bit nums.

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n)$$

[solve: $T(n) = \Theta(n^2)$]

$$AB = (A_h 2^{n/2} + A_l)(B_h \cdot 2^{n/2} + B_l)$$
$$= A_h B_h 2^n + (A_h B_l + A_l B_h) 2^{n/2} + A_l B_l$$

Compute $P := (A_h + A_l)(B_h + B_l)$ (rec. call) $\approx \frac{n}{2}$ bits for each factor

$$P = A_h B_h + A_h B_l + A_l B_h + A_l B_l$$
$$AB = A_h B_h 2^n + (P - A_h B_h - A_l B_l) 2^{n/2} + A_l B_l$$

rec. call, rec. call

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n)$$

[solve:]
$$T(n) = \Theta\left(n^{\lg 3}\right)$$

versus
$$\Theta(n^2) = \Theta\left(n^{\lg 4}\right)$$