ECS 122A

Lecture 2

Today

Review of key terms

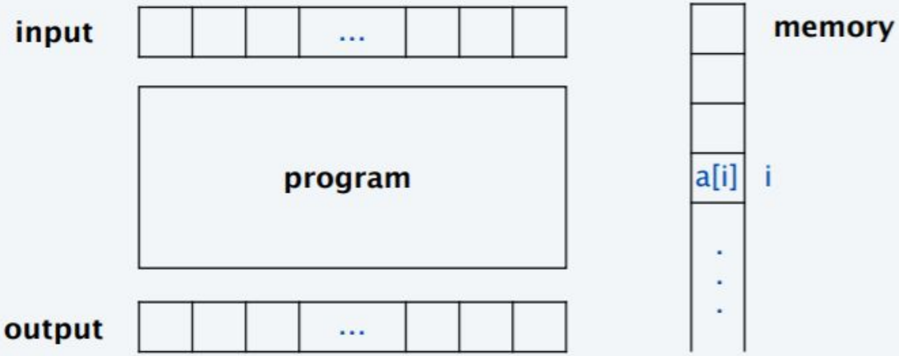
Look at recurrence relations

Word RAM.

assume $w \geq \log_2 n$

- Each memory location and input/output cell stores a w-bit integer.
- Primitive operations: arithmetic/logic operations, read/write memory, array indexing, following a pointer, conditional branch, ...

constant-time C-style operations ($w = 64$)



Running time. Number of primitive operations.

Memory. Number of memory cells utilized.

Desirable scaling property. When the input size doubles, the algorithm should slow down by at most some constant factor C.

Def. An algorithm is poly-time if the above scaling property holds.

There exist constants $c > 0$ and $d > 0$ such that, for every input of size n, the running time of the algorithm is bounded above by $c\,n^d$ primitive computational steps. ⟵ choose $C = 2^d$

Worst-case analysis

Worst case. Running time guarantee for any input of size n.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Other types of analyses

Probabilistic. Expected running time of a randomized algorithm.

Ex. The expected number of compares to quicksort n elements is $\sim 2n \ln n$.

Amortized. Worst-case running time for any sequence of n operations.

Growth	**Terminology**
$O(1)$	constant growth
$O(\log n)$	logarithmic growth
$O(\log^k n)$, for some $k \geq 1$	polylogarithmic growth
$o(n)$	sublinear growth
$O(n)$	linear growth
$O(n \log n)$	log-linear growth
$O(n \log^k n)$, for some $k \geq 1$	polylog-linear growth
$O(n^k)$ for some $k \geq 1$	polynomial growth
$\Omega(n^k)$, for every $k \geq 1$	superpolynomial growth
$\Omega(a^n)$ for some $a > 1$	exponential growth

Trick questions about asymptotic analysis

$f(n) = \log\_3 n$ $g(n) = \log\_2(n)$

$f(n) = O(g(n))$?

$g(n) = O(f(n))$?

Trick questions about asymptotic analysis

f(n) = log_3 n g(n)= log_2(n)

f(n) = O(g(n))?

g(n)= O(f(n))?

$$\log_b(a) = \frac{\log_x(a)}{\log_x(b)}$$

Divide and Conquer

Divide: Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer: the subprob. By solving them recursively (if sub problem is small enough solve it in a straightforward manner)

Combine: combine the solutions of the subproblems into the solution of the original problem.

Recurrences

Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

We can create recurrence to describe the worst case run-times of recursive algorithms that call on themselves with smaller inputs.

For example MergeSort divides the data ½ calls itself and the merges in $\Theta(n)$ time

MergeSort Recurrence

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T(n/2) + \Theta(n), & \text{if } n > 1 \end{cases}$$

Today we will learn how to solve recurrence to provide an asymptotic time bound, using 3 methods

1.) Substitution
2.) Recursion Tree Method
3.) Master Theorem Method

Substitution Method re-examined

The most general method:

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.
3. ***Solve*** for constants.

4.3 Substitution

1.) Guess the form of the solution
2.) Use mathematical induction to find the contents and show the solution works.

Recall to prove f(n)= O(g(n)) we need to find n_0 and constant c such that

f(n) <= c*g(n) for some n>=n_0

Lets prove that T(n)=2T(n/2)+n is O(nlogn)

Lets try ...

$$T(n) = 4T(n/2) + n$$

O(n^2) … We will need to add a lower order constant