Purpose of Design and Analysis of Algorithms

- Purpose of Design: Efficiently solve our problem
- Purpose of Analysis: Predict the behavior of an algorithm without implementing it on a specific computer

How?

- How to measure efficiency?
- How to use the measure to predict (approximately) the behavior?

What does “efficient” mean?

- Economical in <u>time</u> and in <u>space</u>.

  Time is often (but not always) more important these days.
- We count time by counting the number of steps a program goes through to solve a problem, assuming that each step takes no more than a constant time.

Example: I'm thinking of a number between 1 and 1000. Ask me yes/no questions to try to figure it out.

Strategy 1: Is it 1? Is it 2?...

Takes 999 questions in the worst case!

Strategy 2: Is it > 500? If yes, is it > 750? (If no, is it > 250?)...

Takes 10 questions in the worst case!
(Similar to binary search.)

† Not only that, but how many more questions are needed if I can choose between 1 and 2000?

With strategy 1, 1000 more!

With strategy 2, 1 more!

† Incidentally, how would you solve this problem in 10 steps without feedback until you asked all 10?

- We try to relate time to the size of the problem. In this case, time is the number of questions. Size is the size of the interval.
- For strategy 1, $T(n) = n$.
- For strategy 2, $T(n) = \log_2(n)$.
- Doubling problem size doubles the time for strategy 1, but only increases time by one for strategy 2.

How to relate time to the size of a problem?

- Very often, the running time depends not on exact input but only on the *size* of the input, e.g., mergesort.
- When running time is really a function of the particular input, not just of the size of the input, e.g. binary search tree operations:
  - † *worst case:* the maximum, over all inputs of size n, of the running time on the input
  - † *average case:* the average, over all inputs of size n, of the running time on the input.
- We usually use worst case.

  Average case is hard to determine and also depends on the distribution of inputs

Order of Magnitude Notation

- Suppose we analyzed an algorithm for a particular machine and a new one is wheeled in that is faster up to three times. We would then have to redo our analysis.
- Instead we choose to be lazy and ignore factors of 3, 5, indeed any constant.
- When we say the time for an algorithm is $O(\log n)$ ("big oh of log n" or "oh of log n") we could mean it takes time 30 $\log(n)$, 9999 $\log(n)$, and so on.
- For real applications, the constant does matter, but the order of magnitude notation is a good first cut, and usually lets us choose the right algorithm.

Example: 1000 n versus 0.0001 n^2

- For any input smaller than 10^7, second algorithm (0.0001 n^2) is faster. But eventually, the first algorithm is faster.

$\boxed{O(\ )\text{ notation}}$

Capture intuition as follows:

Let $T(n)$ be the time of a particular algorithm for input of size n.

$T(n) = O(g(n))$ iff

$\exists\ c > 0, \exists\ n_0 > 0$ such that $\forall\ n \geq n_0$,

$$T(n) \leq c \cdot g(n).$$

- We don't care what constant is on $g(n)$, as long as for values greater than n_0, $c \cdot g(n)$ dominates $T(n)$. Therefore, $T(n)$ grows no faster than $g(n)$.

- c – lets us ignore constant terms

- n_0 – lets us ignore a few particular values.

Example

The function $T(n) = 3n^3 + 2n^2$ is $O(n^3)$.

Let $n_0 = 1$ and $c = 5$.

For $n \geq n_0$ we have $3n^3 + 2n^2 \leq 5n^3$

- We could also say that $T(n)$ is $O(n^4)$ since we can let $n_0 = 1$, $c = 5$ and for $n \geq 0$ we have $3n^3 + 2n^2 \leq 5n^4$.

  This is a weaker statement than saying it is $O(n^3)$.

Example

Prove 3^n is not $O(2^n)$.

Proof: Suppose there were constants n_0 and c such that for all $n \geq n_0$

$$3^n \leq c \cdot 2^n.$$

Then, $c \geq (3/2)^n$ for all $n \geq n_0$.

But $(3/2)^n$ gets arbitrarily large as n gets larger.

So no constant c can exceed $(3/2)^n$ for all $n \geq n_0$ $\square$

Some properties of $O(\ )$

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

  † $S(n) + T(n) = O(f(n) + g(n))$.

  † $S(n) + T(n) = O(\max(f(n), g(n)))$.

- $\lg n = O(n^{\alpha}), \quad \alpha > 0$

  (any logarithmic function grows slower than a polynomial function)

- $n^k = O(2^n)$

  (any polynomial function grows slower than an exponential function.)

Warning

$O(\ )$ does not always make sense

- Run only a few times (writing, debugging dominates)
- Only on small input
- Time efficient uses too much space
- Accuracy and stability are more important

$\Omega(\ )$ notation

Sometimes, we also know that $T(n)$ grows no slower than a certain function $g(n)$. (Note: in worst case or average case)

Then we say $T(n) = \Omega(g(n))$ (Omega of $g(n)$)

- More formally,

  Let $T(n)$ be the time of an algorithm for input of size n.

  $T(n) = \Omega(g(n))$ iff

  $\exists\ c > 0, \exists\ n_0 > 0$ such that $\forall\ n \geq n_0$,

$$T(n) \geq c \cdot g(n).$$

- Note the similarity between $O(\ )$ and $\Omega(\ )$ notations.
- $O(\ )$ corresponds, loosely, to upper bound.
- $\Omega(\ )$ corresponds, loosely, to lower bound.
- Note: there is a better definition for $\Omega(\ \ )$.

$\boxed{\Theta(\ )\text{ notation}}$

- If $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$ then $T(n) = \Theta(f(n))$

- $T(n) = \Theta(f(n))$, iff
  $\exists\ c_2 > c_1 > 0, \exists\ n_0 > 0$ such that $\forall\ n \geq n_0$,
$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n).$$

- If $T(n) = \Theta(f(n))$ we say $T(n)$ and $f(n)$ are of the same order of magnitude.

- Example: in mergesort $T(n) = \Theta(n \cdot \text{log} n)$.

- The O, Ω, Θ correspond loosely to $\leq, \geq$ and $=$.

- Read textbook for more precise meanings of $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- Read textbook for $o(\ )$ and $\omega(\ )$ notations.

$\boxed{\text{Example}}$

Insertion Sort

– Array $[1, \ldots, n]$ of n elements

– Goal: $A[1]$ is the smallest, $A[2]$ is the second smallest and so on.

– Assume $A[1, \ldots i]$ are sorted.
To insert $A[i+1]$ we need at most i comparisons and $i+1$ data movement.

– This implies $T(n) \leq c \cdot n^2$ which means $T(n) = O(n^2)$

$\boxed{\text{Example}}$

Insertion sort

– Array $[1, \ldots, n]$ of n elements

– Goal: $A[1]$ is the smallest, $A[2]$ is the second smallest and so on.

– *Input:* $A[n]$ is smallest, $A[n-1]$ is second smallest and so on.

– Steps by Insertion Sort: $n(n-1)/2$

– $\Omega(n^2)$ for insertion sort

We already know $T(n) = O(n^2)$

Conclusion: $T(n) = \Theta(n^2)$

Lower bound for a problem

Sometime we can prove lower bound for a *problem* (not for an algorithm); then any algorithm for this problem has that lower bound.

- Worst case sorting by comparison is $\Omega(n \cdot \log n)$.
- Worst case search in an sorted array is $\Omega(\log n)$.
- Worst case search in an unsorted array is $\Omega(n)$.
- Usually it is hard to show lower bound for a problem.
- Meaning:
  - † This is the best (within a constant)
  - † You can quit searching for a better one!

- Occasionally it is easy.
  - † Searching in unsorted arrays.

    Theorem 1. *The lower bound of time complexity for searching a value in an unsorted array of size n is $\Omega(n)$.*

    Proof. Need to check every element at least once. $\square$
  - † What does this mean?

    Once you find an algorithm with time complexity $O(n)$ for searching in unsorted arrays, you stop looking for a better algorithm (if the constant ratio is not your concern)!

Summary

- Running time (time complexity) = number of steps.

- Relate running time to the size of the input:

  † Worst case

  † Average case

- Upper bound $O(\ )$: $\quad T(n) = O(f(n))$
  means $T(n) \leq c \cdot f(n)$ for some $c > 0$ and $n \geq n_0$

- Lower bound $\Omega(\ )$: $\quad T(n) = \Omega(f(n))$
  means $T(n) \geq c \cdot f(n)$ for some $c > 0$ and $n \geq n_0$

- $\Theta(\ )$: upper bound equals lower bound