Algorithm efficiency

- during design phase we often need to pick between different possible algorithms for parts of our program
- runtime efficiency and memory use are important factors
- before we actually write the code, how can we realistically judge how fast it will run or how much memory it will use?
- some estimation/approximation techniques are available, dividing different algorithms into broad classes of efficiency
- in 161: focus mostly on run time, and at an informal level
- in 260: more formal analysis of predicted run time and memory utilization
- in 265: measuring (profiling) actual performance of completed product

Analysis framework

- want to judge how long a program will take to run using algorithm X vs algorithm Y, but there are complications:
  - perhaps X is better with small sets of data but Y is better with larger sets
  - perhaps X is better on average, but is worse in some extreme cases
  - perhaps which one is better depends on specific properties of the input data (e.g. is it nearly sorted?)
- we'll decide much in advance
  - exactly what we're trying to estimate (e.g. average vs worst case),
  - how we'll approximate the size of the input data
  - what kind of algorithm “steps” to count when estimating run time

Best, average, or worst case?

- what is our biggest concern:
  - worst case scenario for each algorithm
  - overall average behaviour for each algorithm
  - best possible scenario for each algorithm
- pick one, and base our computations on that
- average-case or worst-case are the ones most commonly examined
- if you hear people discussing ***"big-O"*** of an algorithm they're talking about worst case analysis

Size of input, N

- how long a program takes to run depends on the size of the problem it is running on (e.g. sorting an array of 100 elements will be quicker than sorting an array of 100,000)
- for any given problem we need to come up with some way to describe the “size” of the input, e.g.:
  - sorting: N may be the number of items to be sorted
  - primality testing: N may be the number of digits in the number being tested
  - image compression: N may be the number of bytes of data to be compressed

Steps to count

- we want to come up with a formula that approximates how many computational steps an algorithm will use to solve a problem of size N
- for worst case behaviour we refer to ***O(N)***, big-O of N
- pick specific set of actions/steps to count, to give ok approximation but allow a simple(ish) expression

Examples of steps to count

- what we count depends very much on the nature of the problem being studied:
  - for sorting algorithms we might count number of times values are compared to one another
  - for computational algorithms we might count number of multiplications/divisions performed
  - for data transfer algorithms we might count number of reads/writes

Classes, or orders of algorithm

- further simplification: will regard algorithms as in same category if they're within a constant factor of each other,
- e.g. algorithm X takes 100N + 7 steps, algorithm Y takes 23N + 904 steps, treat X and Y as same class/order of efficiency
- idea is that if algorithms are within a constant factor of one another then other factors (e.g. maintainability) may be more important
- we're looking for differences so big that the constants aren't relevant, e.g. comparing an $O(N^2)$ vs an $O(N^3)$ algorithm:
  - suppose the $O(N^2)$ is actually kN^2 for some constant k, and the N^3 is just N^3 ... dividing both sides by N^2 shows that for any $N > k$ the N^2 algorithm is still faster
- similarly for an algorithm that is actually kNlogN vs an N^2
  - dividing both sides by N reveals the NlogN is still faster when $N > k \log N$

Growth rates: common orders

(the constants don't matter if you're in a different order!)

N	$\log_2(N)$	sqrt(N)	N log(N)	N^2	N^3	2^N
16	4	4	64	256	4096	65536
256	8	16	2048	65536	16777216	$\sim 10^{77}$
65536	16	256	1048576	$\sim 4.3 \times 10^{9}$	$\sim 2.8 \times 10^{14}$	$\sim 10^{20000}$
1048576	20	1024	20971520	$\sim 2.2 \times 10^{12}$	$\sim 1.2 \times 10^{18}$	$\sim 10^{315000}$
16777216	24	4096	402653184	$\sim 2.8 \times 10^{14}$	$\sim 4.7 \times 10^{21}$	$\sim 10^{5000000}$
268435456	28	16384	7516192768	$\sim 7.2 \times 10^{16}$	$\sim 1.9 \times 10^{25}$	$\sim 10^{80000000}$

Example: searching algorithms

- suppose we want to analyze worst case behaviour of two searching algorithms (e.g. linear search and binary search)
- for size of problem we could use the number of values we to search through
- for operations to be counted we could use the number of comparisons we make between our “target” value and the values we're searching through

linear search: worst case

- worst case:
  - O(N)
  - happens if item is in last spot we check or item is not present, so we have to check all N items
- average case:
  - still O(N) (we wind up looking at half, i.e. 0.5N items)
- best case:
  - O(1), we find it right away

binary search: worst case

- worst case
  - item not present or find at bottom of recursion
  - O(log N) since at most 1 + logN layers of recursion
- best case
  - O(1), we find it right away
- average case
  - also turns out to be O(log N), just getting there with one fewer recursive calls on average

analysis of sorting algorithms

- suppose we decide to analyze worst case behaviour and average case behaviour of different sorting algorithms
- might express size of problem as the number of values to be sorted
- might express operations to count as the number comparisons we perform between different values

bubblesort analysis

- assuming simple bubblesort
  - make O(N) passes through the array
  - each pass we compare current to O(N) elements
  - thus $O(N^2)$ worst case
- if we use smarter bubblesort (quit once we recognize it's already sorted, optimize # passes and how many elements we look at)
  - best case improves to O(N)
  - worst/average cases still $O(N^2)$
  - examine N elements 1st pass, N-1 2nd, ..., 1 on final
  - N-1 + N-2 + ... + 1, rearrange summation in pairs from ends:
  - (N) + (N-1 + 1) + (N-2 + 2) ... each pair is N, N/2 pairs
  - total thus (N)(N/2), and thus $O(N^2)$

Other simple sorts

- similar issues with insertion sort, selection sort
- with algorithm tweaks, best cases can be reduced to O(N):
  - have insert sort outer loop scan forward while next value > current value (go as far as you can while list so far is sorted)
  - have select sort inner loop scan backwards from end while values are decreasing, remember where previous pass left off, can detect when “rest of array” is sorted and we can quit
- average case, worst case still involve $O(N^2)$
  - as with bubblesort, N + N-1 + N-2 + ... + 1, thus $O(N^2)$

mergesort analysis

- each layer of recursion we do merges across all N elements, thus O(N) per layer of merging
  - top layer merging 1 pair, each of N/2 elements
  - next layer merging 2 pairs, each of N/4 elements
  - etc: (# pairs) * (elements per pair) = N
- always O(log N) layers of recursion
  - size of pairs is N/2, N/4, N/8, N/16, until down to 1
    - i.e. $N/2^1$, $N/2^2$, $N/2^3$, ... $N/2^i$ where $i = \log_2 N$
- thus best, average,and worst cases are always O(N logN)

quicksort analysis

- each level of recursion winds up partitioning all N elements, thus O(N) efficiency for each partitioning step
- recursion depth: depends on how good we are at picking pivots
  - best case: O(log2N) levels of recursion, dividing array segments in half each time, like mergesort, thus O(N logN)
  - worst case: (bad pivot choices), if each pivot winds up at the end of it's array segment then the recursive calls deal with N-1 elements, N-2 elements, N-3 elements, ... 1 element,
    - thus similar to earlier computations N+(N-1)+(N-2)+...+1, once again giving $O(N^2)$