2. DIVIDE-AND-CONQUER

Raveen de Silva, r.desilva@unsw.edu.au
office: K17 202
Course Admin: Anahita Namvar, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 3, 2021

Table of Contents

Rates of growth

- You should be familiar with true-ish statements such as:

  Heap sort is faster than bubble sort.
  Linear search is slower than binary search.

- We would like to make such statements more precise.
- We also want to understand when they are wrong, and why it matters.

Asymptotic notation

- We need a way to compare two functions, in this case representing the runtime of each algorithm. However, comparing values directly is surprisingly fraught:
  - Outliers
  - Implementation details
- We prefer to talk in terms of asymptotics.
  - For example, if the size of the input doubles, the function value could (approximately) double, quadruple, etc.
  - A function which quadruples will eventually ‘beat’ a function which doubles, in any circumstances.

"Big-Oh" notation

Definition

We say $f(n) = O(g(n))$ if

> *there exist positive constants C and N such that $0 \leq f(n) \leq C\, g(n)$ for all $n \geq N$.*

- $g(n)$ is said to be an *asymptotic upper bound* for $f(n)$.
- Informally, $f(n)$ is eventually (i.e. for large n) controlled by a multiple of $g(n)$, i.e. $f(n)$ grows "no slower than $g(n)$".
- Useful to (over-)estimate the complexity of a particular algorithm, e.g. insertion sort runs in $O(n^2)$ time.

Big Omega notation

Definition

We say $f(n) = \Omega(g(n))$ if

> *there exist positive constants c and N such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq N$.*

- $g(n)$ is said to be an *asymptotic lower bound* for $f(n)$.
- Informally, $f(n)$ eventually (i.e. for large n) dominates a multiple of $g(n)$, i.e. $f(n)$ grows “no faster than $g(n)$”.
- Useful to say that any algorithm solving a particular problem runs in at least $\Omega(g(n))$, e.g. any comparison sort runs in $\Omega(n \log n)$.

Landau notation

- $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$.
- There are strict versions of Big-Oh and Big Omega notations: namely little-oh (o) and little omega (ω) respectively.

Definition

We say $f(n) = \Theta(g(n))$ if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

- $f(n)$ and $g(n)$ are said to have the same asymptotic growth rate.

Properties of Landau notation

Sum property

If $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 + f_2 = O(g_1 + g_2)$.

Product property

If $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 \cdot f_2 = O(g_1 \cdot g_2)$.

In particular, if $f = O(g)$ and λ is a constant, then $\lambda \cdot f = O(g)$ also.

The same properties hold if O is replaced by Ω, Θ, o or ω.

Logarithms

Definition

For $a, b > 0$ and $a \neq 1$, let $n = \log_a b$ if $a^n = b$.

Properties

$$a^{\log_a n} = n$$

$$\log_a(mn) = \log_a m + \log_a n$$

$$\log_a(n^k) = k \log_a n$$

Change of Base Rule

Theorem

For $a, b, x > 0$ and $a, b \neq 1$, we have

$$\log_a x = \frac{\log_b x}{\log_b a}.$$

- The denominator is constant with respect to x!
- Therefore $\log_a n = \Theta(\log_b n)$, that is, logarithms of any base are interchangeable in asymptotic notation.
- We typically write $\log n$ instead.

Table of Contents

An old puzzle

Problem

We are given 27 coins of the same denomination; we know that one of them is counterfeit and that it is lighter than the others. Find the counterfeit coin by weighing coins on a pan balance only three times.

Hint

You can reduce the search space by a third in one weighing!

An old puzzle

Solution

- Divide the coins into three groups of nine, say A, B and C.
- Weigh group A against group B.
  - If one group is lighter than the other, it contains the counterfeit coin.
  - If instead both groups have equal weight, then group C contains the counterfeit coin!
- Repeat with three groups of three, then three groups of one.

Divide and Conquer

- This method is called “divide-and-conquer”.
- We have already seen a prototypical “serious” algorithm designed using such a method: the MERGE-SORT.
- We split the array into two, sort the two parts recursively and then merge the two sorted arrays.
- We now look at a closely related but more interesting problem of counting inversions in an array.

Counting the number of inversions

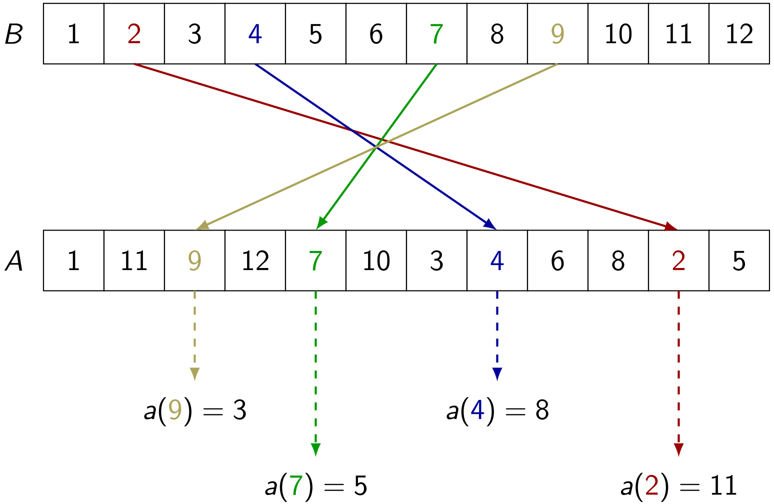
- Assume that you have m users ranking the same set of n movies. You want to determine for any two users A and B how similar their tastes are (for example, in order to make a recommender system).

- How should we measure the degree of similarity of two users A and B?

- Lets enumerate the movies on the ranking list of user B by assigning to the top choice of user B index 1, assign to his second choice index 2 and so on.

- For the i^{th} movie on B's list we can now look at the position (i.e., index) of that movie on A's list, denoted by $a(i)$.

Counting the number of inversions



Counting the number of inversions

- A good measure of how different these two users are, is the total number of *inversions*, i.e., total number of pairs of movies i, j such that movie i precedes movie j on B's list but movie j is higher up on A's list than the movie i.

- In other words, we count the number of pairs of movies i, j such that $i < j$ (movie i precedes movie j on $B's$ list) but $a(i) > a(j)$ (movie i follows movie j on A's list, in positions $a(i)$ and $a(j)$ respectively).

- For example 1 and 2 do not form an inversion because

$$1 = a(1) < a(2) = 11,$$

but 4 and 7 do form an inversion because

$$5 = a(7) < a(4) = 8.$$

Counting the number of inversions

- An easy way to count the total number of inversions between two lists is to test each pair $i < j$ of movies on one list, and add one to the total if they are inverted in the second list. Unfortunately this produces a quadratic time algorithm, $T(n) = \Theta(n^2)$.

- We now show that this can be done in a much more efficient way, in time $O(n \log n)$, by applying a divide-and-conquer strategy.

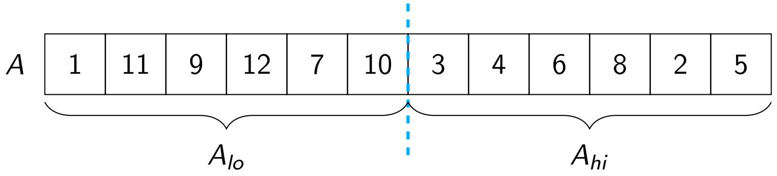
- Clearly, since the total number of pairs is quadratic in n, we cannot afford to inspect all possible pairs.

- The main idea is to tweak the MERGE-SORT algorithm, by extending it to recursively both sort an array A **and** determine the number of inversions in A.

Counting the number of inversions

- We split the array A into two (approximately) equal parts $A_{lo} = A[1..m]$ and $A_{hi} = A[m+1..n]$, where $m = \lfloor n/2 \rfloor$.

- Note that the total number of inversions in array A is equal to the sum of the number of inversions $I(A_{lo})$ in A_{lo} (such as 9 and 7) plus the number of inversions $I(A_{hi})$ in A_{hi} (such as 4 and 2) plus the number of inversions $I(A_{lo}, A_{hi})$ across the two halves (such as 7 and 4).



Counting the number of inversions

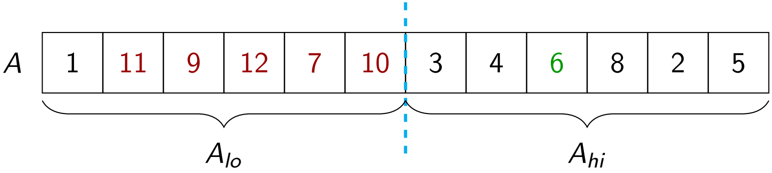
- We have

$$I(A) = I(A_{lo}) + I(A_{hi}) + I(A_{lo}, A_{hi}).$$

- The first two terms of the right-hand side are the number of inversions within A_{lo} and within A_{hi}, which can be calculated recursively.
- It seems that the main challenge is to evaluate the last time, which requires us to count the inversions which cross the partition between the two sub-arrays.

Counting the number of inversions

- In our example, how many inversions involve the 6?



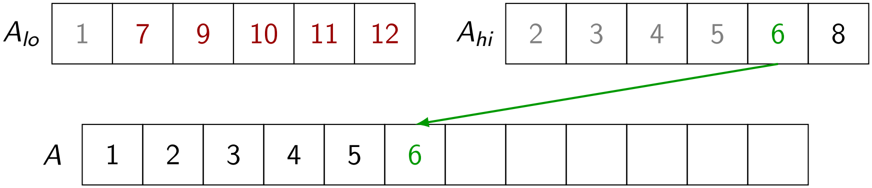
- It's the number of elements of A_{lo} which are greater than 6, but how would one compute this systematically?

- The idea is to not only count inversions across the partition, but also sort the array. We can then assume that the subarrays A_{lo} and A_{hi} are sorted in the process of counting $I(A_{lo})$ and $I(A_{hi})$.

Counting the number of inversions

- We proceed to count $I(A_{lo}, A_{hi})$ (specifically, counting each inversion according to the lesser of its elements) and simultaneously merge as in Merge-Sort.

- Each time we reach an element of A_{hi}, we have inversions with this number and each of the remaining elements in A_{lo}. We therefore add the number of elements remaining in A_{lo} to the answer.



Counting the number of inversions

- On the other hand, when we reach an element of A_{lo}, all inversions involving this number have already been counted.
- We have therefore counted the number of inversions within each subarray ($I(A_{lo})$ and $I(A_{hi})$) as well as the number of inversions across the partition (($I(A_{lo}, A_{hi})$), and adding these gives $I(A)$ as required.
- Clearly this algorithm has the same complexity as MERGE-SORT, i.e. $\Theta(n \log n)$.
- **Next:** we look to generalise this method of divide and conquer.

Recurrences

- Recurrences are important to us because they arise in estimations of time complexity of divide-and-conquer algorithms.

- Recall that counting inversions in an array A of size n required us to:

  - recurse on each half of the array (A_{lo} and A_{hi}), and

  - count inversions across the partition, in linear time.

- Therefore the runtime $T(n)$ satisfies

$$T(n) = 2T\left(\frac{n}{2}\right) + c\,n.$$

Recurrences

- Let $a \geq 1$ be an integer and $b > 1$ a real number, and suppose that a divide-and-conquer algorithm:
  - reduces a problem of size n to a many problems of smaller size n/b,
  - with overhead cost of $f(n)$ to split up the problem and combine the solutions from these smaller problems.
- The time complexity of such an algorithm satisfies

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n).$$

Recurrences

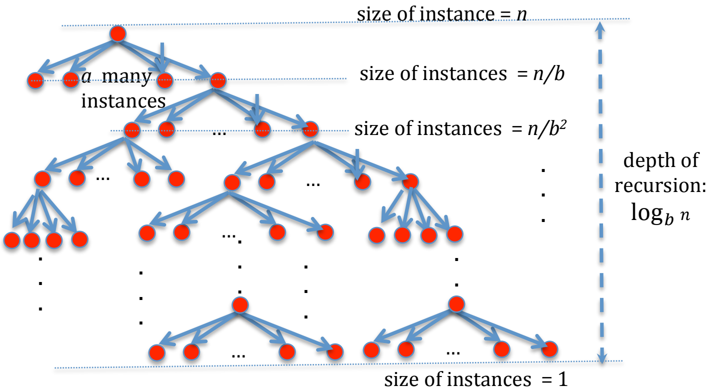
Note

Technically, we should be writing

$$T(n) = a\,T\left(\left\lceil\frac{n}{b}\right\rceil\right) + f(n)$$

but it can be shown that the same asymptotics are achieved if we ignore the integer parts and additive constants.

Recurrences of the form $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$


size of instance = n
a many instances
size of instances = n/b
size of instances = n/b^2
depth of recursion: $\log_b n$
size of instances = 1

Solving Recurrences

- Some recurrences can be solved explicitly, but this tends to be tricky.
- Fortunately, to estimate the efficiency of an algorithm we **do not** need the exact solution of a recurrence
- We only need to find:
  - the **growth rate** of the solution i.e., its asymptotic behaviour, and
  - the (approximate) **sizes of the constants** involved (more about that later)
- This is what the **Master Theorem** provides (when it is applicable).

Table of Contents

A puzzle

Problem

Five pirates have to split 100 bars of gold. They all line up and proceed as follows:

- The first pirate in line gets to propose a way to split up the gold (for example: everyone gets 20 bars)
- The pirates, including the one who proposed, vote on whether to accept the proposal. If the proposal is rejected, the pirate who made the proposal is killed.
- The next pirate in line then makes his proposal, and the 4 pirates vote again. If the vote is tied (2 vs 2) then the proposing pirate is still killed. Only majority can accept a proposal.

This process continues until a proposal is accepted or there is only one pirate left.

A puzzle

Problem (continued)

Assume that every pirate has the same priorities, in the following order:

1. not having to walk the plank;
2. getting as much gold as possible;
3. seeing other pirates walk the plank, just for fun.

A puzzle

Problem (continued)

What proposal should the first pirate make?

Hint

Assume first that there are only two pirates, and see what happens. Then assume that there are three pirates and that they have figured out what happens if there were only two pirates and try to see what they would do. Further, assume that there are four pirates and that they have figured out what happens if there were only three pirates, try to see what they would do. Finally assume there are five pirates and that they have figured out what happens if there were only four pirates.

That's All, Folks!!