دانشگاه آزاد اسلامی واحد تبریز

نام درس: طراحی الگوریتم ها

بخش:

Substitution method, and randomized algorithms

نام استاد: دکتر مسعود کارگر

If you think lectures are too slow

- You are not alone.
- I'll try to put fun problems on the side of slides for you to think about.
  - (Also you can find all the typos in my slides and email them to me) ☺

Note: even if you don't think lectures are too slow, you can go back and look at these problems afterwards!

Are there functions f(n) and g(n) that are both increasing, but so that f(n) is neither O(g(n)) nor Ω(g(n))?

Ollie the Over-achieving Ostrich

Let’s get a move-on…

- Last time: we saw a cool (and complex!) recursive algorithm for solving SELECT.

- One idea: Use MergeSort and take the k’th smallest.
  - Time O(n log(n)). Can we do better??
- Idea: pick a pivot that’s close to the median, and recurse on either side of the pivot.
- Cool trick: Use recursion to also pick the pivot!
- CLAIM: This runs in time O(n).

Last time we ended up with this:

- $T(n) \leq c \cdot n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10} + 5\right)$

The cn is the O(n) work done at each level for PARTITION

The T(n/5) is for the recursive call to get the median in FINDPIVOT

The T(7n/10 + 5) is for the recursive call to SELECT for either L or R.

- How can we solve this?
- The sub-problems don't have the same size.
  - The master method doesn't work.
  - Recursion trees get complicated.
- The substitution method gives us a way.
  - fancy "guess-and-check"

Try solving this using a recursion tree!

Ollie the over-achieving ostrich

being sloppy about floors and ceilings!

The substitution method (by example)

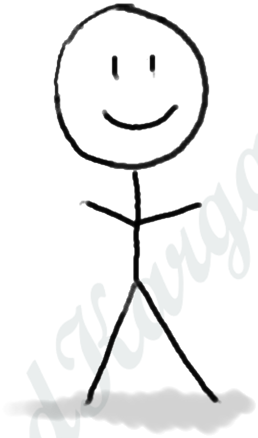
- example: $T(n) \leq 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$,
  - with T(n) = 10n for n < 10.
- First, make a guess about the answer.
- Check your guess using induction.
  - Suppose that your guess holds for all k < n.
  - $T(n) \leq 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$
  - $T(n) \leq 3n + 10\left(\frac{n}{5}\right) + 10\left(\frac{n}{2}\right)$
  - $T(n) \leq 3n + 2n + 5n = 10n.$
  - This establishes the inductive hypothesis for n.

  - (And the base case is satisfied: $T(n) \leq 10n$ for n < 10.)
- So T(n) = O(n).

This is not the same as our SELECT example; we'll come back to that.

Inductive hypothesis:

I think $T(k) \leq 10k.$



How did we come up with that hypothesis?

- Doesn’t matter for the correctness of the argument, but..
  - Be very lucky.
  - Play around with the recurrence relation to try to get an idea before you start.
  - Start with a hypothesis with a variable in it, and try to solve for that variable at the end.

Example of how to come up with a guess.

- First, make a guess about what the correct term should be: but leave a variable "C" in it, to be determined later.
- example: $T(n) \leq 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$,
  - with T(n) = 10n for n < 10.
- Check your guess using induction.
  - Suppose that your guess holds for all k < n.
  - $T(n) \leq 3n + T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right)$
  - $T(n) \leq 3n + C\left(\frac{n}{5}\right) + C\left(\frac{n}{2}\right)$
  - $T(n) \leq 3n + \frac{Cn}{5} + \frac{Cn}{2}$.
  - If I want that to be Cn, then I can solve for C…

Inductive hypothesis:

I think $T(n) \leq Cn$.

Back to SELECT

The cn is the O(n) work done at each level for PARTITION

- $T(n) \leq c \cdot n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10} + 5\right)$

The T(7n/10 + 5) is for the recursive call to SELECT for either L or R.

The T(n/5) is for the recursive call to get the median in FINDPIVOT

- Inductive hypothesis (aka our guess):
- $T(n) \leq \begin{cases} d \cdot 100 & if \;\; n \leq 100 \\ d \cdot n & if \;\; n > 100 \end{cases}$ (aka, T(n) = O(n)).

for d = 20c.

How on earth did we come up with this? Try to arrive at this guess on your own.

Ollie the over-achieving ostrich

Finally, let's prove we can do SELECT in time O(n)

- Base case:
  - If n <= 50, we can assume our alg. takes time <= 50d.
    - *(You should justify: WHY IS THIS OKAY?)*
- Inductive step: Suppose (*) holds for all sizes k < n. Then
- $T(n) \leq c \cdot n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10} + 5\right)$

$$\leq c \cdot n + d \cdot \frac{n}{5} + d \cdot \left(\frac{7n}{10} + 5\right)$$

$$\leq n \left(c + \frac{d}{5} + \frac{7d}{10}\right) + 5d$$

$$\leq n \left(c + \frac{20c}{5} + \frac{140 \cdot c}{10}\right) + 100\, c$$

$$= (19\, n + 100\,)c$$

$\leq 20c \cdot n$ whenever n > 100.

$$= \ d \cdot n$$

$$(*) \quad T(k) \leq \begin{cases} d \cdot 100 & if \ \ k \leq 100 \\ d \cdot k & if \ k > 100 \end{cases}$$

for d = 20c.

Here come some computations: no need to pay too much attention, just know that you **can** do these computations.

This is pretty pedantic! But it's worth being careful about the constants when doing inductive arguments. (see: your homework).

Nearly there!

- By induction, the inductive hypothesis (*) applies for all n.
- Termination: Observe that this is exactly what we wanted to show!
  - There exists:
    - a constant d>0 (which depends on the constant c from the running time of PARTITION…)
    - an n_0 (aka 101)
  - so that for all n >= n_0, T(n) <= d n.
  - By definition, T(n) = O(n).
  - Hooray!

$$(*)\quad T(n) \leq \begin{cases} d \cdot 100 & if\ \ n \leq 100 \\ d \cdot n & if\ \ n > 100 \end{cases}$$

for d = 20c.

- Conclusion:

We can implement SELECT in time O(n).

Quick recap before we move on

- We can do SELECT (in particular, MEDIAN) in time O(n).
- We analyzed this with the substitution method.

Next up:

- Randomized algorithms.

Randomized algorithms

- The algorithm gets to use randomness.
- It should always be correct (for this class).
- But the runtime can be a random variable.

- We'll see a few randomized algorithms for sorting.
  - BogoSort
  - QuickSort
- BogoSort is a pedagogical tool.
- QuickSort is important to know. (in contrast with BogoSort…)

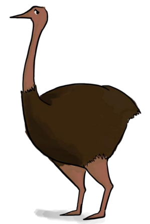
Example of a randomized sorting algorithm

- BogoSort(A):
  - **While** true:
    - Randomly permute A.
    - Check if A is sorted.
    - **If** A is sorted, **return** A.

- This algorithm is always correct:
  - If it returns, then it returns a sorted list.

- Informal Runtime Analysis (and probability refresher):
  - E[runtime] = ?
  - Pr[randomly permuted array is sorted] = ?
    - 1/n!
  - We expect to permute A n! times before it's sorted.
  - E[runtime] = $O(n \cdot n!)$ = BIG.
  - Worst-case runtime?
    - Infinity!

Suppose that you can draw a random integer in {1,…,n} in time O(1). How would you randomly permute an array in-place in time O(n)?



Ollie the over-achieving ostrich

We expect to roll a 6-sided die 6 times before we see a 1. We expect to flip a fair coin twice before we see heads.

Worst case means that an adversary chooses the randomness.

Example of a better randomized algorithm: QuickSort

- Runs in expected time O(nlog(n)).
- Worst-case runtime $O(n^2)$.
- Easier to implement than MergeSort, and the constant factors inside the O() are very small.
- In practice often more desirable.

Quicksort

We want to sort this array.

7	6	3	5	1	2	4

First, pick a "pivot."
Do it at random.

random pivot!

Next, partition the array into "bigger than 5" or "less than 5"

This PARTITION step takes time O(n). (Notice that we don't sort each half). [same as in SELECT]

Arrange them like so:

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Recurse on L and R:

1	2	3	4	5	6	7

PseudoPseudoCode for what we just saw

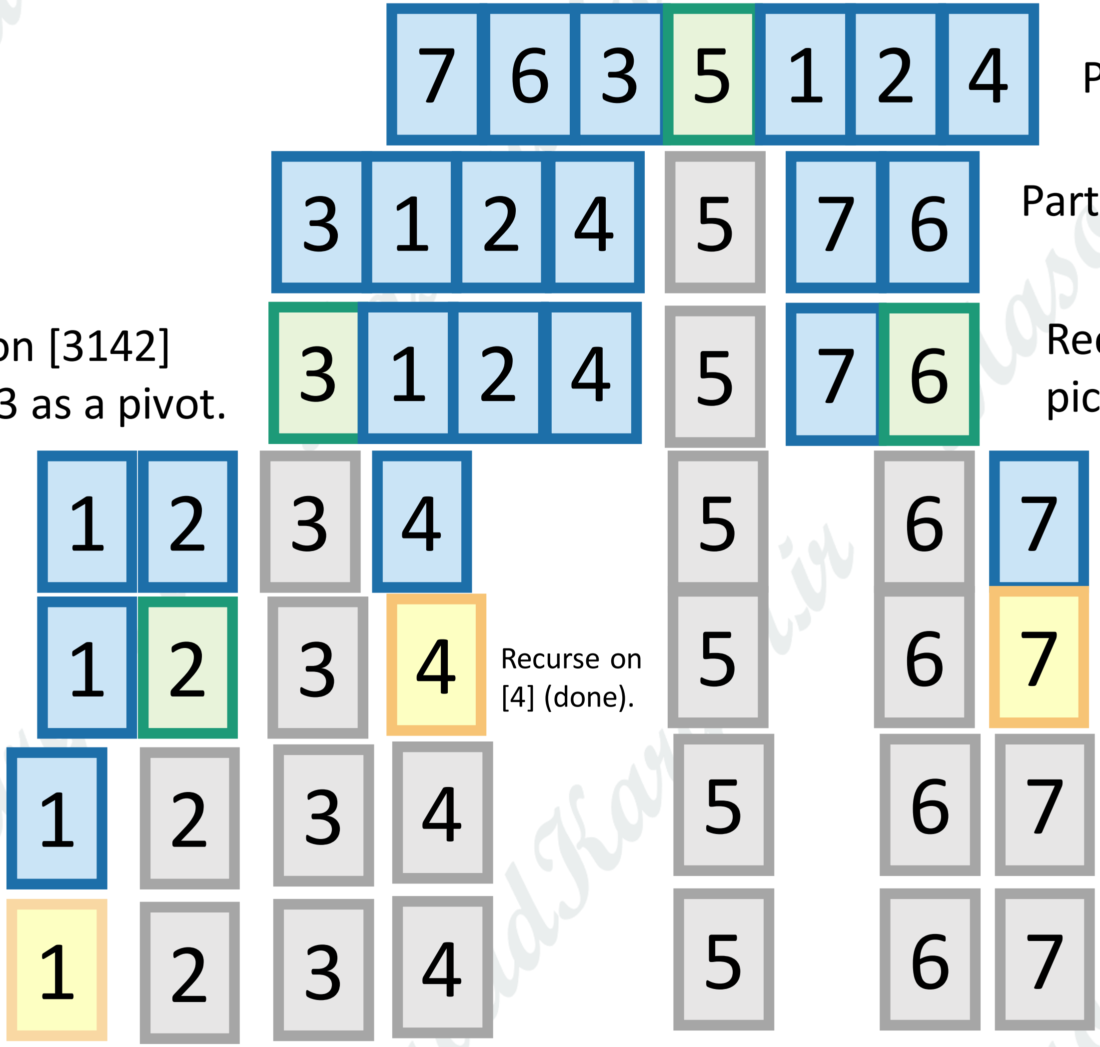
See CLRS for more detailed pseudocode.

- QuickSort(A):
  - **If** len(A) <= 1:
    - **return**
  - Pick some x = A[i] at random. Call this the pivot.
  - PARTITION the rest of A into:
    - L (less than x) and
    - R (greater than x)
  - Replace A with [L, x, R] (that is, rearrange A in this order)
  - QuickSort(L)
  - QuickSort(R)

How would you do all this in-place in time O(n)?

Ollie the over-achieving ostrich

Example of recursive calls



Pick 5 as a pivot

Partition on either side of 5

Recurse on [3142] and pick 3 as a pivot.

Recurse on [76] and pick 6 as a pivot.

Partition around 3.

Partition on either side of 6

Recurse on [12] and pick 2 as a pivot.

Recurse on [7], it has size 1 so we're done.

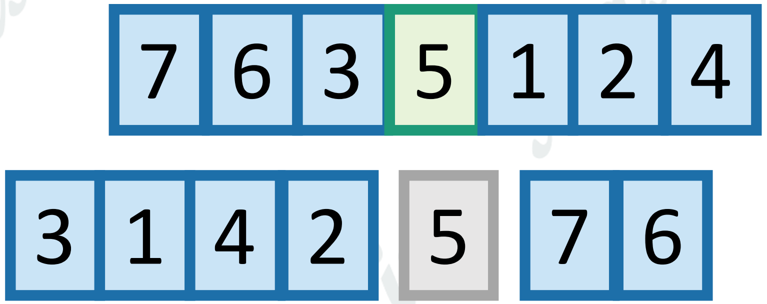
partition around 2.

17

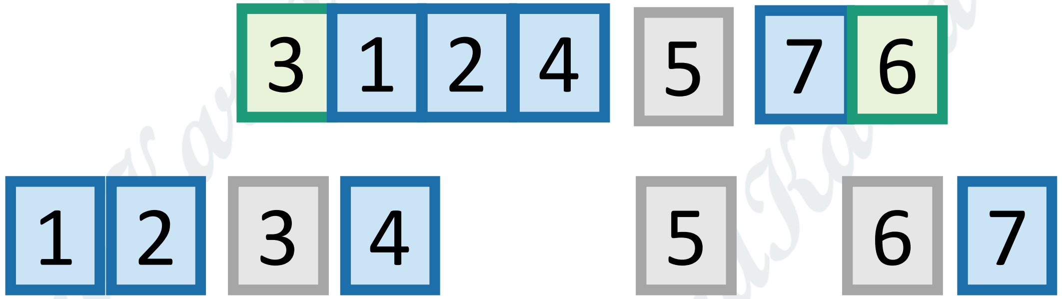
Recurse on [1] (done).

How long does this take to run?

- We will count the number of comparisons that the algorithm does.
  - This turns out to give us a good idea of the runtime. (Not obvious).
- How many times are any two items compared?



In the example before, everything was compared to 5 once in the first step....and never again.



But not everything was compared to 3.
5 was, and so were 1,2 and 4.
But not 6 or 7.

Each pair of items is compared either 0 or 1 times. Which is it?

7	6	3	5	1	2	4

Let's assume that the numbers in the array are actually the numbers 1,…,n

Of course this doesn't have to be the case! It's a good exercise to convince yourself that the analysis will still go through without this assumption. (Or see CLRS)

- Whether or not a,b are compared is a random variable, that depends on the choice of pivots. Let's say

$$X_{a,b} = \begin{cases} 1 & \textit{if a and b are ever compared} \\ 0 & \textit{if a and b are never compared} \end{cases}$$

- In the previous example $X_{1,5} = 1$, because item 1 and item 5 were compared.
- But $X_{3,6} = 0$, because item 3 and item 6 were NOT compared.
- Both of these depended on our random choice of pivot!

Counting comparisons

- The number of comparisons total during the algorithm is

$$\sum_{a=1}^{n} \sum_{b=a+1}^{n} X_{a,b}$$

- The expected number of comparisons is

$$E\left[\sum_{a=1}^{n} \sum_{b=a+1}^{n} X_{a,b}\right] = \sum_{a=1}^{n} \sum_{b=a+1}^{n} E[X_{a,b}]$$

using linearity of expectations.

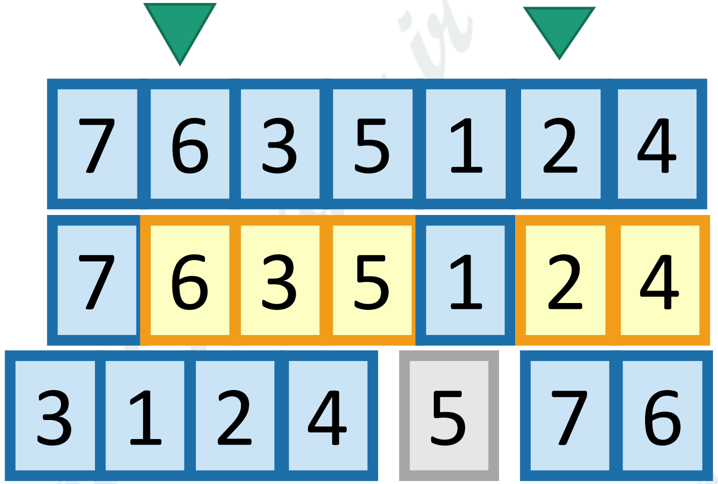
Counting comparisons

- So we just need to figure out $E[\ X_{a,b}\ ]$
- $E[\ X_{a,b}\ ] = P(\ X_{a,b} = 1\ )\cdot 1 + P(\ X_{a,b} = 0\ )\cdot 0 = P(X_{a,b} = 1)$
  - (using definition of expectation)
- So we need to figure out

$P(X_{a,b} = 1)$ = the probability that a and b are ever compared.

expected number of comparisons:

$$\sum_{a=1}^{n}\sum_{b=a+1}^{n} E[\ X_{a,b}]$$



Say that a = 2 and b = 6. What is the probability that 2 and 6 are ever compared?

This is exactly the probability that either 2 or 6 is first picked to be a pivot out of the highlighted entries.

If, say, 5 were picked first, then 2 and 6 would be separated and never see each other again.

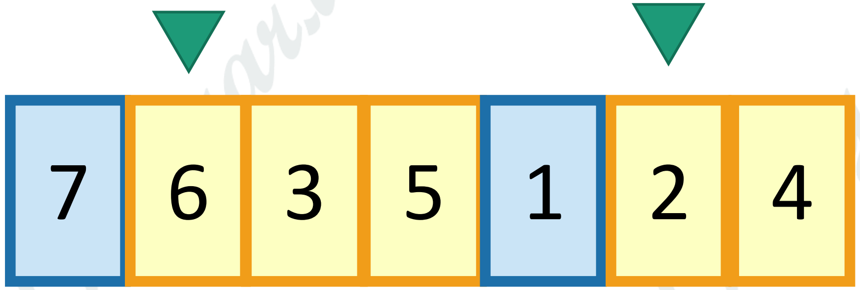
Counting comparisons

$$P(\,X_{a,b} = 1\,)$$

= probability a,b are ever compared

= probability that one of a,b are picked first out of all of the b – a +1 numbers between them.

$$= \frac{2}{b - a + 1}$$



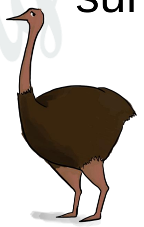
All together now…

Expected number of comparisons

- $E\left[\sum_{a=1}^{n}\sum_{b=a+1}^{n} X_{a,b}\right]$ — This is the expected number of comparisons throughout the algorithm
- $= \sum_{a=1}^{n}\sum_{b=a+1}^{n} E[\,X_{a,b}]$ — linearity of expectation
- $= \sum_{a=1}^{n}\sum_{b=a+1}^{n} P(\,X_{a,b} = 1)$ — definition of expectation
- $= \sum_{a=1}^{n}\sum_{b=a+1}^{n} \frac{2}{b-a+1}$ — the reasoning we just did

- This is a big nasty sum, but we can do it.
- We get that this is less than 2n ln(n).

Do this sum!



Ollie the over-achieving ostrich

Are we done?

- We saw that E[number of comparisons] = O(n log(n))
- Is that the same as E[running time]?

- In this case, **yes**.
- We need to argue that the running time is dominated by the time to do comparisons.
- (See CLRS for details).

- QuickSort(A):
  - **If** len(A) <= 1:
    - **return**
  - Pick some x = A[i] at random. Call this the pivot.
  - PARTITION the rest of A into:
    - L (less than x) and
    - R (greater than x)
  - Replace A with [L, x, R] (that is, rearrange A in this order)
  - QuickSort(L)
  - QuickSort(R)

Worst-case running time for QuickSort
(if time)

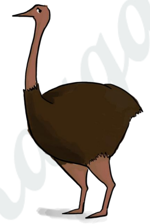
- Suppose that an adversary is choosing the random pivots for you.
- Then the running time might be $O(n^2)$ [on board]
- In practice, this doesn't usually happen.

- **Aside**: We worked really hard last week to get a deterministic algorithm for SELECT, by picking the pivot very cleverly.
- What happens if you pick the pivot randomly?
- Turns out this is also usually a good idea.

Recap

- We can do SELECT and MEDIAN in time O(n).
- We already knew how to sort in time O(nlog(n)) with MergeSort.
- The randomized algorithm QuickSort also runs in expected time O(nlog(n)).
- In practice, QuickSort is often nicer.

Code up both QuickSort and MergeSort. Which is more of a headache? And which runs faster?



Ollie the over-achieving ostrich

- Skills of today:
  - substitution method
  - analysis of randomized algorithms.

Next time

- Could we sort faster than O(n log(n))??

قدردانی