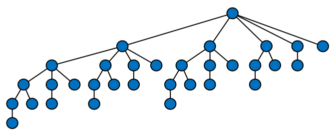
Lecture 6. Divide and Conquer Continued, Selection

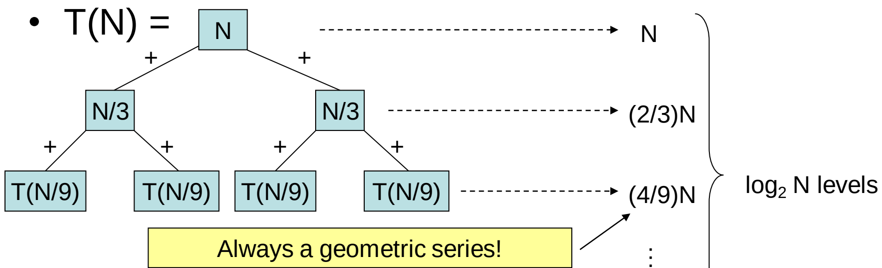
CpSc 8400: Algorithms and Data Structures
Brian C. Dean



School of Computing
Clemson University
Spring, 2016

Recall: Tree Expansions for Solving Recurrences

- Consider T(N) = aT(n/b) + f(n);   e.g. T(N) = 2T(N/3) + Θ(N)
- T(N) =



- Solution is:
  - Decreasing: solution T(n) = Θ(root contribution)
  - Increasing: solution T(n) = Θ(leaf contribution) = Θ(n^p), p = $\log_b$ a
  - Unchanging: solution T(n) = Θ(L log n), where L is the contribution on each of the log n levels.

Extra Log Factors

- Suppose our "f(n)" term has an extra logarithmic factor: $T(n) = aT(n/b) + n^{\alpha} \log^{\beta} n$.
- Here, $T(n)$ =

$\Theta(n^{\alpha} \log^{\beta} n)$	if $\alpha > p$	(decreasing series)
$\Theta(n^{\alpha} \log^{\beta+1} n)$	if $\alpha = p$	(unchanging series)
$\Theta(n^{p})$	if $\alpha < p$	(increasing series)

- For the purpose of using a tree expansion to determine if the series is increasing, decreasing, or unchanging, it's ok to temporarily ignore the existence of the $\log^{\beta} n$ term.

3

Practice

- $T(n) = T(2n/3) + \Theta(1)$
- $T(n) = 4T(n/2) + \Theta(n^2)$
- $T(n) = 3T(n/3) + \Theta(n\sqrt{n})$
- $T(n) = 6T(n/3) + \Theta(n^2 \log^3 n)$
- $T(n) = 7T(n/8) + \Theta(n \log^5 n)$
- $T(n) = 17T(n/15) + \Theta(n \log^{500} n)$
- $T(n) = T(n/2) + \Theta(\log n)$
- $T(n) = 14T(n/14) + n \log^2 n - 17n + 5$

4

More Sophisticated Example: Stable In-Place Sorting

Algorithm	Runtime	Stable	In-Place?
Bubble Sort	$O(n^2)$	Yes	Yes
Insertion Sort	$O(n^2)$	Yes	Yes
Merge Sort	$\Theta(n \log n)$	Yes	No
Randomized Quicksort	$\Theta(n \log n)$ w/high prob.	No*	Yes*
Deterministic Quicksort	$\Theta(n \log n)$	No*	Yes*
Heap Sort	$\Theta(n \log n)$	No*	Yes*

- Is stable in-place sorting possible in $O(n \log n)$ time? Yes, but very complicated.
- However, there are simple $O(n \log^2 n)$ time approaches…

* = can be transformed into a stable, out-of-place algorithm

Warm-Up: In-Place Block Swap

- Suppose we have a length-n array comprised of two blocks A and B of potentially different size:

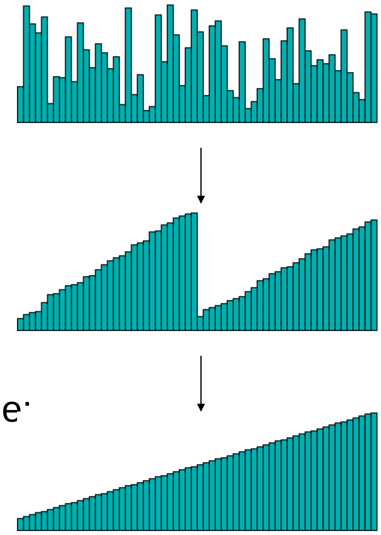
- How can we rearrange this array **in-place** in $\Theta(n)$ time so that its contents are now:

- Note: this is easy if A and B have the same size…

Stable In-Place Merge Sort

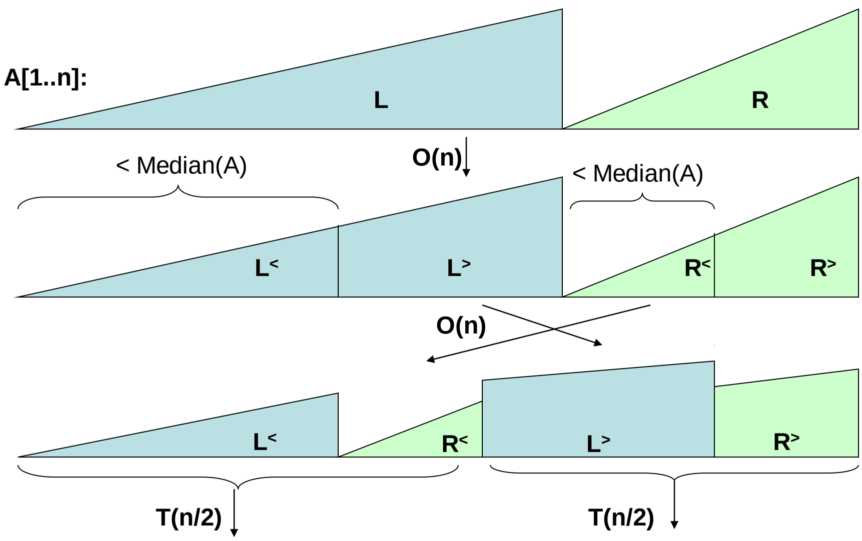
- The standard $\Theta(n)$ merging algorithm is stable but not in-place.
- We'll show how to merge in $\Theta(n \log n)$ time in a stable and in-place fashion.
- When used with merge sort, this gives a $\Theta(n \log^2 n)$ runtime:

$$T(n) = 2T(n/2) + \Theta(n \log n)$$



7

$\Theta(n \log n)$ Stable In-Place Merge

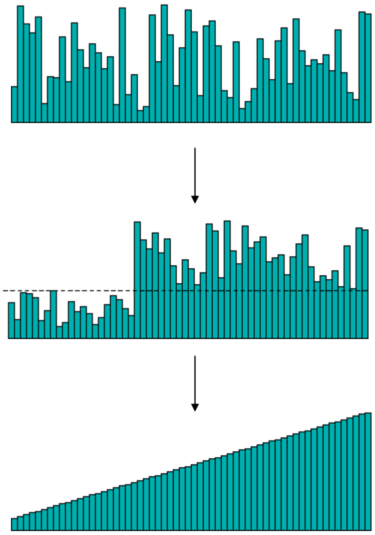


8

Stable In-Place Quicksort

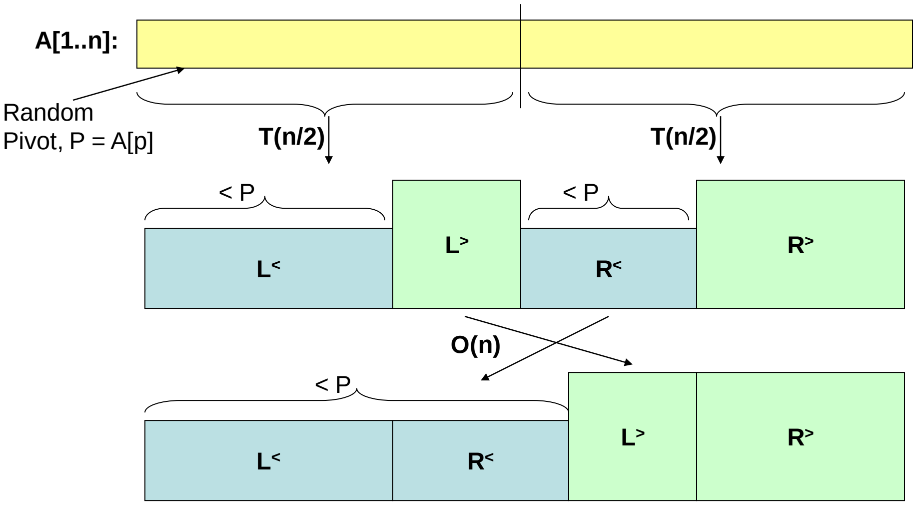
- The standard $\Theta(n)$ partition algorithm is in-place but not stable.
- We'll show how to partition in $\Theta(n \log n)$ time in a stable and in-place fashion.
- When used with randomized quicksort, this gives a $\Theta(n \log^2 n)$ runtime with high probability.

  Note: hard to derandomize!



9

$\Theta(n \log n)$ Stable In-Place Partition



10

In-Place Matrix Transposition

- Consider an n x m matrix stored in row-major order (row 1, followed by row 2, etc.)
- We'd like to transpose the matrix **in-place**, so it becomes stored in column-major order.
- On our homework, we showed how to do this in O(N log N) time, where N = mn.

11

Special Case: Finding the Transpose of an m x 2 Matrix

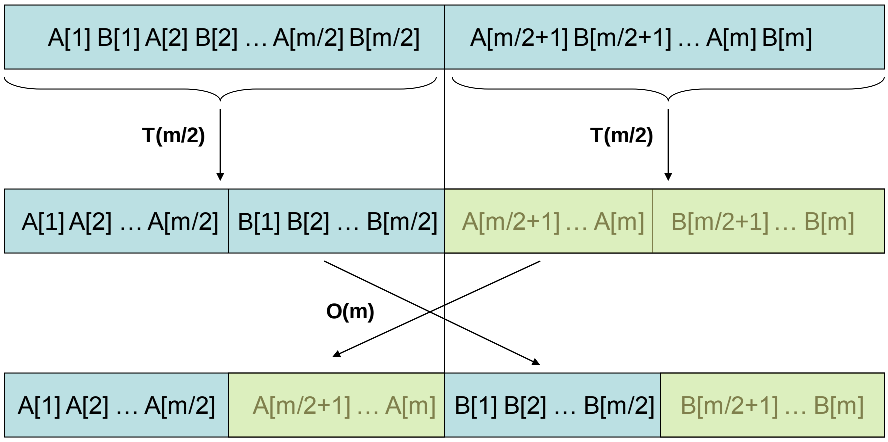
- Let A[1..m] and B[1..m] denote the contents of the two columns of our matrix.
- In memory, we start with the matrix stored as:

  A[1] B[1] A[2] B[2] A[3] B[3] … A[m] B[m]
- We would like to rearrange it so it's stored as:

  A[1] A[2] A[3] … A[m] B[1] B[2] B[3] … B[m]
- This is the inverse of a **perfect shuffle** permutation.
- … and it's easy to perform in-place in Θ(m log m) time using a simple divide-and-conquer algorithm, similar to those we use for stable in-place sorting.

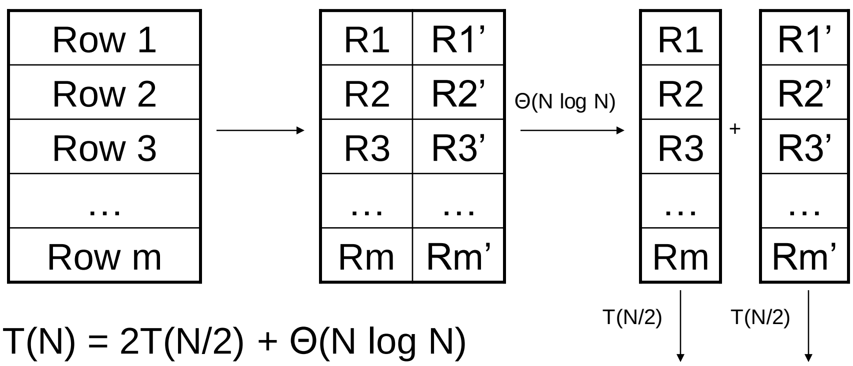
12

In-Place Inverse Perfect Shuffle



General m x n Matrices

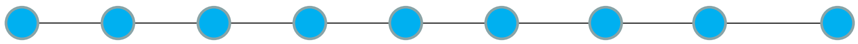
- Now that we can perform an in-place inverse perfect shuffle in $\Theta(n \log n)$ time, we can use this as a building block to devise a simple $\Theta(N \log^2 N)$ algorithm for in-place matrix transposition!



$T(N) = 2T(N/2) + \Theta(N \log N)$

The Firing Squad Problem

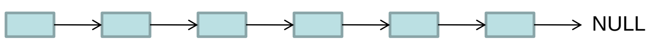
- N parallel processors hooked together in a line:



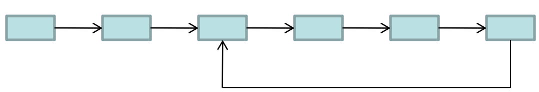
- Each processor doesn't know N, and only has a constant number of bits of memory (so it can't even count to N).
- Processors synchronized to a global clock. In each time step, a processor can:
  - Perform some simple calculation.
  - Exchange messages with its neighbors.
- At some point in time, we give the leftmost processor a "ready!" message.
- Sometime in the future, we want all the processors to enter the same state "fire!" all in the same time step.

Linked List Ending in a Loop

- You are given a pointer to the beginning of a linked list.
- It either ends by pointing to NULL:



or it ends by pointing back into itself, forming a loop:



- **Goal:** Determine as quickly as possible which of these two cases is occurring.
- **To make things interesting:** You aren't allowed to modify the list, or to use substantial amounts of extra memory…

Selection

- **Selection** is the problem of locating the kth largest element in an unsorted array / linked list.
- The kth largest element is also called the kth **order statistic**.
- Common values of k:
  - k = 1: minimum
  - k = n: maximum
  - k = n/2: median
- It's easy to find the min and max in $\Theta(n)$ time.
- For the median, we can easily achieve $\Theta(n \log n)$ time by first sorting the array, but can we do it faster?

"QuickSelect"

- To select for the item of **rank** k in an array A[1..n].
- As in quicksort, pick a pivot element and partition A in linear time:

Elements < pivot	pivot	Elements > pivot

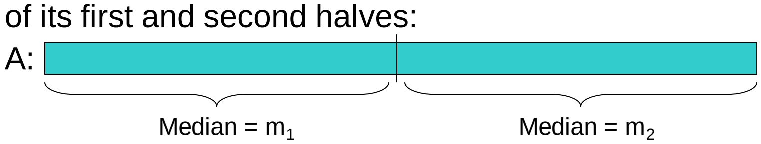
- After partitioning, the pivot element ends up being placed where it would in the sorted ordering of A
  – So we know the rank, r, of the pivot!
  – If k = r, the pivot is the element we seek and we're done.
  – If k < r, select for the element of rank k on the left side.
  – If k > r, select for the element of rank k – r on the right side.
- Just like quicksort, except we only recurse on one subproblem instead of both.

QuickSelect: Choosing a Pivot

- As in quicksort, the difficulty with quickselect is choosing a good pivot.
- The ideal choice would be the median element, but then again we're trying to compute the median with this algorithm!
- Good choice: choose a random element as the pivot.
  – Intuition: "on average" we split the problem into two reasonably large pieces. And if we always manage to split into reasonably large pieces, we're solving a recurrence like $T(n) = T(n/2) + \Theta(n)$ or $T(n) = T(9n/10) + \Theta(n)$, the solution of which is $T(n) = \Theta(n)$.
  – In a few weeks, we'll show that the **expected** running time of randomized quickselect is $\Theta(n)$, even though the worse case is still $\Theta(n^2)$, same as quicksort.

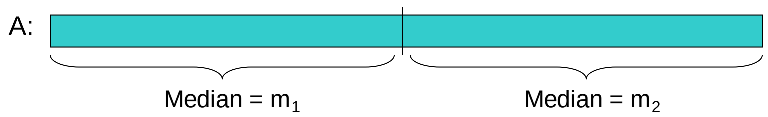
Deterministic Selection

- Is there a $\Theta(n)$ deterministic selection algorithm?
  – This fundamental problem remained open for some time before finally being resolved (positively) in the early 1970s.
- How might we solve this using divide and conquer…
  – Focus on finding the median (note that if we can find the median in linear time, we can select for any other order statistic also in linear time).
  – Take an array A[1..n] and recursively compute the median of its first and second halves:



  – What can we say about the median of the entire array A compared to m_1 and m_2?

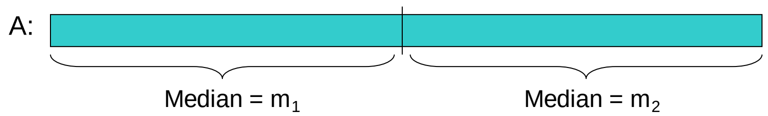
Deterministic Selection



- Claim: The median of A lies between min(m_1, m_2) and max(m_1, m_2).

21

Deterministic Selection

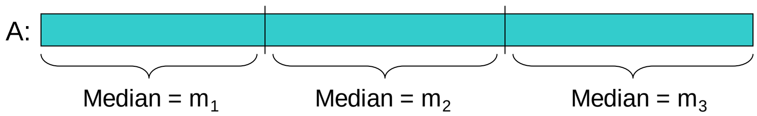


- Claim: The median of A lies between min(m_1, m_2) and max(m_1, m_2).
- For the block with the smaller median, discard all elements < the median. For the block with the larger median, discard all elements > the median.
  - Since we discard n/4 elements < median(A) and n/4 elements > median(A), the median of the leftover elements is still the median of A!
  - So now recursively select for the median of the remaining n/2 elements.
  - But what is the total running time? $T(n) = 3T(n/2) + \Theta(n)$.

22

Deterministic Selection, Take 2…

- Ok, so the obvious "divide in half" approach didn't give a fast enough running time. What if we divide into 3 pieces?

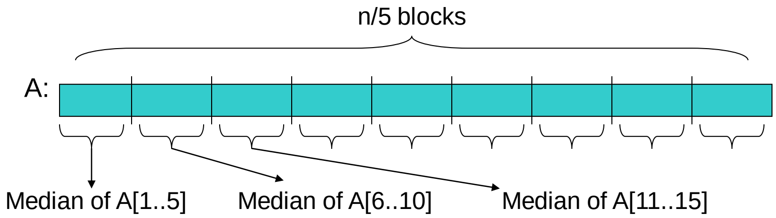


- Sadly, this doesn't work either…

23

Deterministic Selection, Take 3…

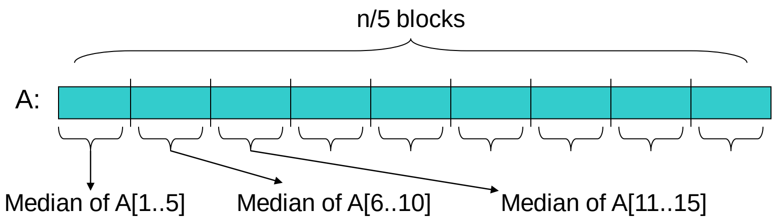
- But remarkably, this works:



- Divide A into n/5 blocks of length 5 and find the median of each block (only Θ(n) time!).
- Now recursively find the median M of the block medians (T(n/5) time). This turns out to be a sufficiently good choice for the pivot for quickselect.

24

Deterministic Selection



- M = median of block medians.
- Claim: at least 3/10 of the elements of A are ≤ M, and at least 3/10 of the elements of A are ≥ M.
- So if we use M as a pivot in quickselect, we will recurse on a subproblem of size at most 7n/10.
- Total runtime: $T(n) \leq T(n/5) + T(7n/10) + \Theta(n)$.
  - Decreasing series! Which solves to $T(n) = \Theta(n)$.

Deterministic Selection: Applications

- The median is an ideal partitioning point for divide and conquer algorithms.
- Example: with quicksort, we can now find the median and partition in linear time in the worst case, so this gives us a $\Theta(n \log n)$ deterministic version of quicksort.
  - Can it still be made to operate in place?
  - The hidden constant is slightly large, since the hidden constant in the $\Theta(n)$ runtime for deterministic selection is also a bit large.