Searching and Sorting

Bengt J. Nilsson

Malmö University, Sweden

Today's Content

Searching

- Linear Search
- Binary Search
- Exponential and Binary Search

Sorting

- Selection Sort
- Insertion Sort
- The Priority Queue ADT
- Sorting with Priority Queues
- In Place Sorting — Heap Sort
- Merge Sort
- Quicksort
- A Lower Bound on Sorting

Divide and Conquer Algorithms

Searching: Linear Search

Do you remember this piece of code?

A is an array with n elements, *element* is a variable with a *key* item

```
for i ← 0 to |A| − 1 do
    if element.key = A[i].key then return A[i] endif
endfor
```

the code does *linear search* through an (unordered) array

Searching: Linear Search

Do you remember this piece of code?

A is an array with n elements, *element* is a variable with a *key* item

```
for i ← 0 to |A| − 1 do
    if element.key = A[i].key then return A[i] endif
endfor
```

the code does *linear search* through an (unordered) array

We argued that the running time for this is $O(n)$!

Can you search faster?

Searching: Linear Search

Do you remember this piece of code?

A is an array with n elements, *element* is a variable with a *key* item

```
for i ← 0 to |A| − 1 do
    if element.key = A[i].key then return A[i] endif
endfor
```

the code does *linear search* through an (unordered) array

We argued that the running time for this is $O(n)$!

Can you search faster?

In general NO!

Searching: Binary Search

But if we know that data are organized in a particular way, then **we can do better**! Assume: data is ordered (in increasing) order

Algorithm `binarySearch`
Input: array A, indices i, j in A, and search element *element*

if $i \geq j$ **then**
  if $A[i].key = element.key$ **then return** $A[i]$ **endif**
else
  var $k \leftarrow \lfloor (i+j)/2 \rfloor$, **if** $A[k].key = element.key$ **then return** $A[k]$ **endif**
  if $A[k].key > element.key$ **then**
    return `binarySearch`$(A, i, k-1, element)$
  else
    return `binarySearch`$(A, k+1, j, element)$
  endif
endif

End `binarySearch`

Call: `binarySearch`$(A, 0, |A|-1, element)$

Searching: Binary Search

Complexity: $T(n) \leq T(n/2) + O(1)$

Master theorem gives: $T(n) \in O(\log n)$

Avoid recursion? You can in this case but for $n = 10^9$, $\log_2 n = 30$

Not worth the engineering effort!

Searching: Exponential and Binary Search

Complexity: $T(n) \leq T(n/2) + O(1)$

Master theorem gives: $T(n) \in O(\log n)$

Avoid recursion? You can in this case but for $n = 10^9$, $\log_2 n = 30$

Not worth the engineering effort!

How do we search if we don't know what n is? No upper index given but array is ordered?

var $k \leftarrow 1$
while $k < A.maxsize$ **and** $A[k].key < element.key$ **do** $k \leftarrow 2 \cdot k$ **endwhile**
`binarySearch`$(A, k/2, \min\{k, A.maxsize - 1\}, element)$

Takes $O(\log p)$ time, p is index of searched element (if it exists)

Called **Exponential and Binary Search**!

Sorting

To obtain ordered arrays, we need to know how to order them!

Sorting: "sorting refers to ordering data in an increasing or decreasing manner according to some linear relationship among the data items.

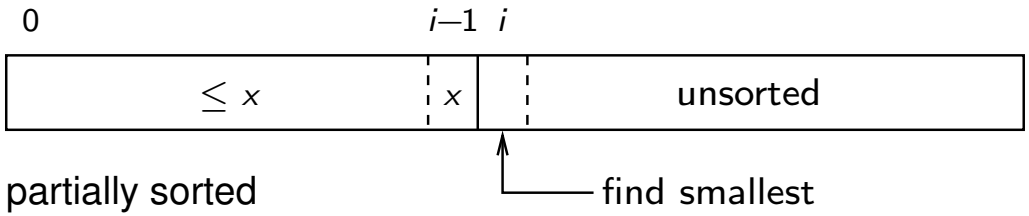
1. ordering: arranging items in a sequence ordered by some criterion
2. categorizing: grouping items with similar properties.

Ordering items is the combination of categorizing them based on equivalent order, and ordering the categories themselves."

"In computer science, arranging in an ordered sequence is called *sorting*. Sorting is a common operation in many applications, and efficient algorithms have been developed to perform it."

Sorting: Selection Sort

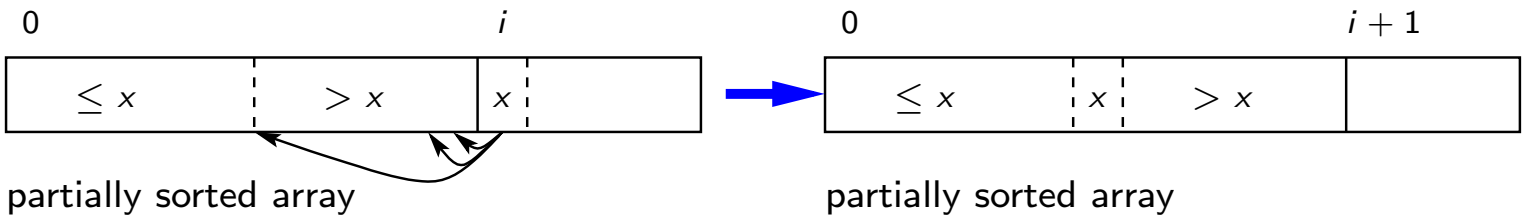
```
for i ← 0 to |A| − 1 do
    var min ← i
    for j ← i + 1 to |A| − 1 do
        if A[j].key < A[min].key then min ← j endif
    endfor
    swap A[i] and A[min]
endfor
```



Complexity:
$\sum_{i=0}^{n-1} O(n-i) = \sum_{k=1}^{n} O(k) = O(n(n+1)/2) = O(n^2)$ time

Sorting: Insertion Sort

```
var i ← 1
while i < |A| do
        var j ← i
        var temp ← A[i]
        while j > 0 and A[j − 1].key > temp.key do
                A[j] ← A[j − 1]
                j ← j − 1
        endwhile
        A[j] ← temp
        i ← i − 1
endwhile
```

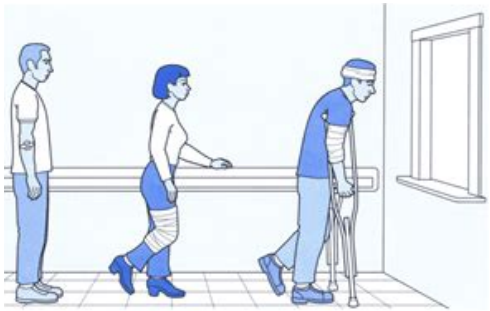


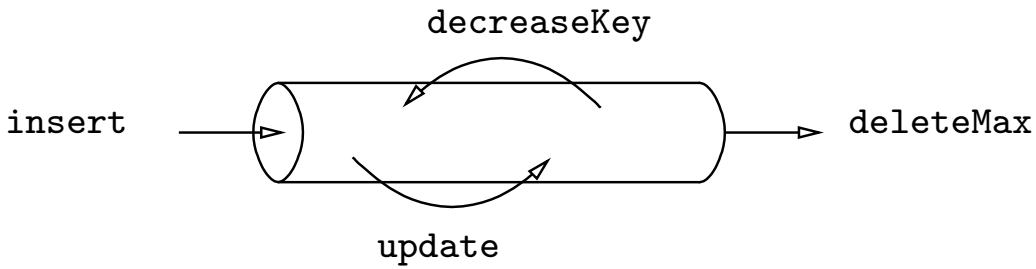
Complexity: $O(n^2)$ time

Faster $O(n)$ time if input already sorted or almost sorted

Sorting: The Priority Queue ADT

Operations: `insert`, `deleteMax` (`deleteMin`), `update` (`decreaseKey`)
Keeps a FIFO structure on priorities and with update possibility





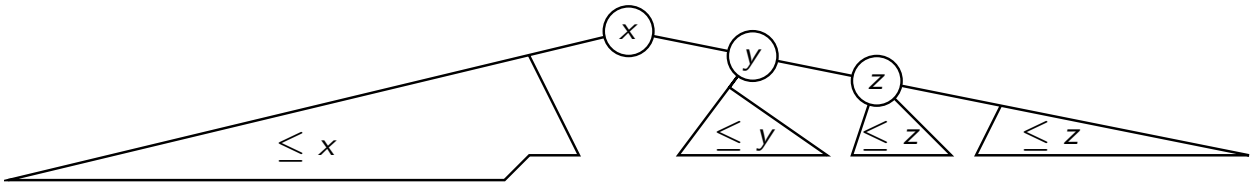
Implementation using array or linked list requires some operations taking $O(n)$ time
Implementation using binary search tree guarantees $O(\log n)$ time per operation
Implementation using Fibonacci heap gives $O(1)$ amortized time for `insert`/`update`

Sorting: The Priority Queue ADT

Can be implemented using a balanced binary search tree, `insert` and updates as usual.
`deleteMin` returns the leftmost value in the tree.

There is a better way: a **heap**

Perfectly balanced binary tree where path along parents from every leaf to the root is in increasing order

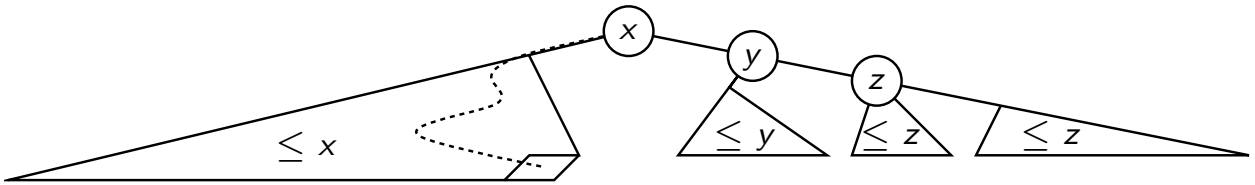


Sorting: The Priority Queue ADT

Can be implemented using a balanced binary search tree, `insert` and updates as usual.
`deleteMin` returns the leftmost value in the tree.

There is a better way: a **heap**

Perfectly balanced binary tree where path along parents from every leaf to the root is in increasing order



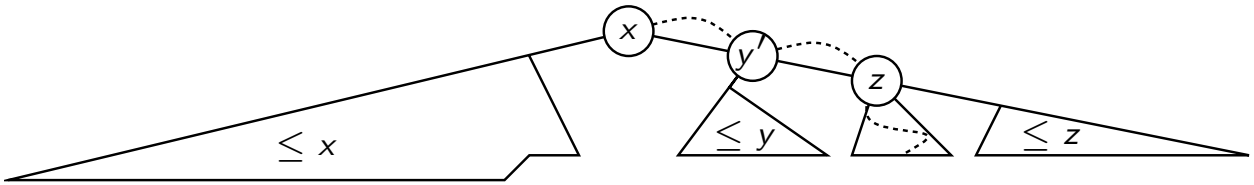
`insert`: add element in last position of perfectly balanced tree. Do `trickle-up` along parent path

Sorting: The Priority Queue ADT

Can be implemented using a balanced binary search tree, `insert` and updates as usual.
`deleteMin` returns the leftmost value in the tree.

There is a better way: a **heap**

Perfectly balanced binary tree where path along parents from every leaf to the root is in increasing order



`insert`: add element in last position of perfectly balanced tree. Do `trickle-up` along parent path

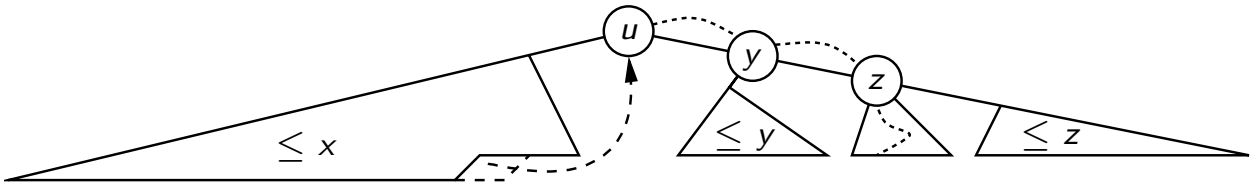
`update`: if key is increased, do `trickle-up` along parent path. If key is decreased, do `trickle-down` along largest child path

Sorting: The Priority Queue ADT

Can be implemented using a balanced binary search tree, `insert` and updates as usual.
`deleteMin` returns the leftmost value in the tree.

There is a better way: a **heap**

Perfectly balanced binary tree where path along parents from every leaf to the root is in increasing order



`insert`: add element in last position of perfectly balanced tree. Do `trickle-up` along parent path

`update`: if key is increased, do `trickle-up` along parent path. If key is decreased, do `trickle-down` along largest child path

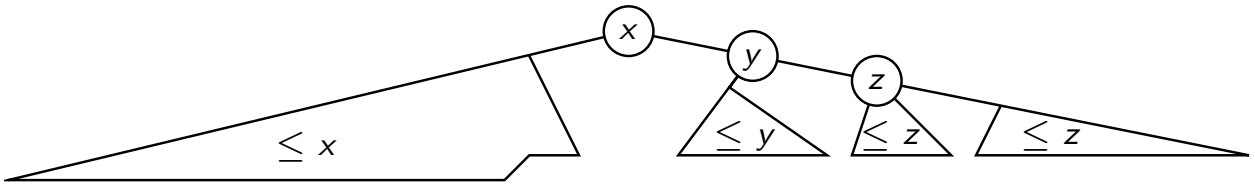
`deleteMax`: place element in last position in root, then do `trickle-down` along largest child path, return original root

Sorting: The Priority Queue ADT

Can be implemented using a balanced binary search tree, `insert` and updates as usual.
`deleteMin` returns the leftmost value in the tree.

There is a better way: a **heap**

Perfectly balanced binary tree where path along parents from every leaf to the root is in increasing order



`insert`: add element in last position of perfectly balanced tree. Do `trickle-up` along parent path

`update`: if key is increased, do `trickle-up` along parent path. If key is decreased, do `trickle-down` along largest child path

`deleteMax`: place element in last position in root, then do `trickle-down` along largest child path, return original root

Let's get back to sorting!

Sorting: Sorting with Priority Queues

Elements assumed to lie in an array A in positions $0 \ldots n-1$.
High level code:

```
var priorityQueueType Q ← createPQ(A)
for i ← n − 1 downto 0 do
    A[i] ← deleteMax(Q)
endfor
```

Simple sorting method

Complexity for sorting $O(|\texttt{createPQ}| + n \cdot |\texttt{deleteMax}|))$

list/array: $O(|\texttt{createPQ}|) = O(n)$, $|\texttt{deleteMax}| = O(n) \Rightarrow O(n^2)$

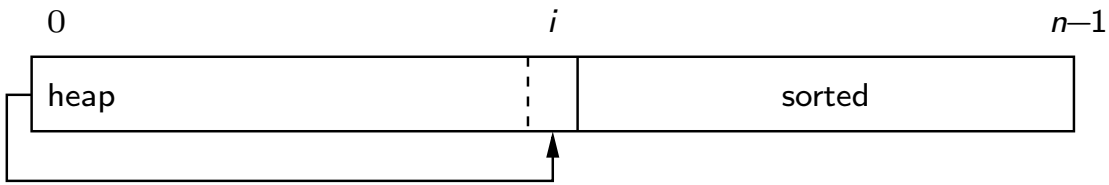
balanced binary search tree: $O(|\texttt{createPQ}|) = O(n \log n)$,
$|\texttt{deleteMax}| = O(\log n) \Rightarrow O(n \log n)$

Benefits of using a heap?

Sorting: Heap Sort — In Place Sorting

Elements assumed to lie in an array A in positions $0 \ldots n-1$.
High level code:

```
createHeap(A)
for i ← n − 1 downto 0 do
    A[i] ← deleteMax(A)
endfor
```



We can do the sorting *in place*. No need for extra data structure, left part of the array serves as heap, right part contains the partially ordered elements.

Complexity for sorting, still: $O(|\texttt{createHeap}| + n \cdot |\texttt{deleteMax}|)$

Sorting: Heap Sort

Algorithm `deleteMax`
Input: A heap A from indices 0 to $n-1$
$min \leftarrow A[0]$, $A[0] \leftarrow A[n-1]$
`trickleDown`$(A, 0, n-2)$
return *min*
End `deleteMax`

Complexity for `deleteMax`, one `trickleDown`: $O(\log n)$

Complexity for Heap sort:
$O(|\texttt{createHeap}| + n \cdot |\texttt{deleteMax}|)) = O(|\texttt{createHeap}| + n \log n)$

Sorting: Heap Sort

Algorithm `deleteMax`
Input: A heap A from indices 0 to $n-1$
$min \leftarrow A[0]$, $A[0] \leftarrow A[n-1]$
`trickleDown`$(A, 0, n-2)$
return *min*
End `deleteMax`

Complexity for `deleteMax`, one `trickleDown`: $O(\log n)$

Complexity for Heap sort:
$O(|\texttt{createHeap}| + n \cdot |\texttt{deleteMax}|)) = O(|\texttt{createHeap}| + n \log n)$

Heap can be created by doing n inserts into the heap, each taking $O(\log n)$ time $\Rightarrow$ $O(n \log n)$ time
We can do slightly better!

Sorting: Heap Sort

Elements assumed to lie in an array A in positions $0 \ldots n-1$.
High level code:

Algorithm `createHeap`
Input: An array A of elements indexed from 0 to $n-1$
for $i \leftarrow n/\!/2$ **downto** 0 **do**
    `trickleDown`$(A, i, n-1)$
endfor
End `createHeap`

Algorithm `trickleDown`
Input: An array A and element at i to move down, k is final index
var $child \leftarrow 2 \cdot i + 1$
if $child < k$ **and** $A[child] < A[child+1]$ **then**
    $child \leftarrow child + 1$
endif
if $child \leq k$ **and** $A[i] < A[child]$ **then**
    swap $A[i]$ and $A[child]$
    `trickleDown`$(A, child, k)$
endif
End `trickleDown`

Complexity: from bottom level to root we do this many `trickleDowns`

$$0 \cdot n/2 + 1 \cdot n/4 + \cdots + (h-1) \cdot n/2^h = \frac{n}{2}\left(\sum_{\lambda=0}^{h-1} \frac{\lambda}{2^\lambda}\right) \in O(n)$$

Sorting: Heap Sort

Complexity for `deleteMax`, one `trickleDown`: $O(\log n)$

Complexity for `createHeap`: $O(n)$

Complexity for Heap sort:
$O(|\texttt{createHeap}| + n \cdot |\texttt{deleteMax}|)) = O(n + n \log n) = O(n \log n)$

Simple, in place, sorting method

Sorting: Merge Sort

Algorithm `mergesort`
Input: array A, i, j indices in A
if $i \geq j$ **then**
  return A
else
  $k \leftarrow \lfloor (i+j)/2 \rfloor$
  return $\texttt{merge}(\texttt{mergesort}(A, i, k), \texttt{mergesort}(A, k+1, j))$
endif
End `mergesort`

Call: $\texttt{mergesort}(A, 0, |A| - 1)$
Complexity: $T(n) = 2T(n/2) + M(n) + O(1) \in ?$

Sorting: Merge Sort

How do you merge?

Algorithm `merge`
Input: two arrays $A[0, \ldots, k-1]$ and $B[0, \ldots, l-1]$
if $|A| = 0$ **then return** B **endif**
if $|B| = 0$ **then return** A **endif**
if $A[0] \leq B[0]$ **then**
  return $A[0] \circ \texttt{merge}(A[1, \ldots, k-1], B[0, \ldots, l-1])$
else
  return $B[0] \circ \texttt{merge}(A[0, \ldots, k-1], B[1, \ldots, l-1])$
endif
End `merge`

Complexity: $n = k + l$

$$M(n) = M(n-1) + O(1) = M(n-2) + O(1) + O(1) = \cdots$$
$$= M(0) + \underbrace{O(1) + \cdots + O(1)}_{n} \in O(n)$$

Sorting: Merge Sort

Unfortunately merge sort uses a lot of extra memory to move data back and forth
Not practically useful for random access data but variants (multiway sort) works well for data stored on tape

Can we avoid all that data shifting?
Quicksort (C.A.R. Hoare, 1961)
Also a D&C algorithm but does its main work before the recursive calls rather than after them

Sorting: Quicksort

Algorithm `quicksort`
Input: array A, i, j indices in A
if $j - i \leq 30$? **then**
  return `bruteForceSort`(A, i, j)
else
  $k \leftarrow$ `partition`(A, i, j)
  `quicksort`$(A, i, k-1)$
  `quicksort`$(A, k+1, j)$
  return A
endif
End `quicksort`

Call: `quicksort`$(A, 0, |A| - 1)$
Complexity: $\Big[k = 0, \ldots, n-1\Big]$

$$T(n) = T(n-k-1) + T(k) + P(n) + O(1) = ?$$

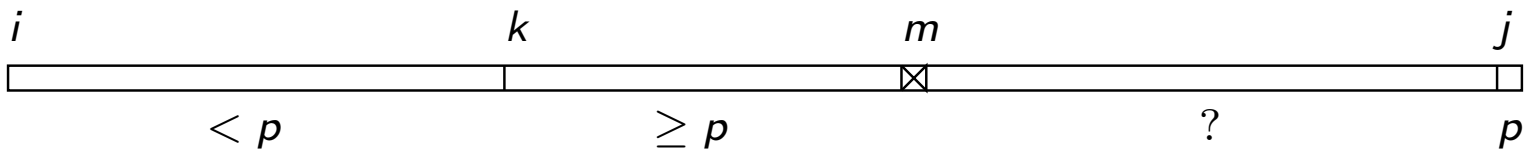
Sorting: Quicksort

Many different pivot selection schemes exist! We will use random

Algorithm `partition`
Input: array A, i, j indices in A

Choose $i \leq l \leq j$ randomly
$p \leftarrow A[l]$, $\texttt{swap}(A, l, j)$, $k \leftarrow i$
for $m \leftarrow i, \ldots, j-1$ **do**
    if $A[m] < p$ **then** $\texttt{swap}(A, m, k)$, $k \leftarrow k+1$ **endif**
endfor
$\texttt{swap}(A, k, j)$, **return** k

End `partition`



Running time: $P(n) \in O(n)$

Sorting: Quicksort

So, $P(n) \in O(n)$ and $\Big[k = 0, \ldots, n-1\Big]$

$$T(n) = T(n-k-1)+T(k)+P(n)+O(1) = T(n-k-1)+T(k)+O(n)$$

Worst case: pivot always becomes end point element

$$\begin{aligned} T(n) &= T(n-1) + O(n) = T(n-2) + O(n) + O(n) = \cdots \\ &= T(0) + \underbrace{O(n) + \cdots + O(n)}_{n} = O(n^2) \end{aligned}$$

Sorting: Quicksort

So, $P(n) \in O(n)$ and $\Big[k = 0, \ldots, n-1\Big]$

$$T(n) = T(n-k-1)+T(k)+P(n)+O(1) = T(n-k-1)+T(k)+O(n)$$

Worst case: pivot always becomes end point element

$$\begin{aligned} T(n) &= T(n-1) + O(n) = T(n-2) + O(n) + O(n) = \cdots \\ &= T(0) + \underbrace{O(n) + \cdots + O(n)}_{n} = O(n^2) \end{aligned}$$

Wrong model!

Our algorithm is randomized $\rightarrow$ worst case is extremely unlikely

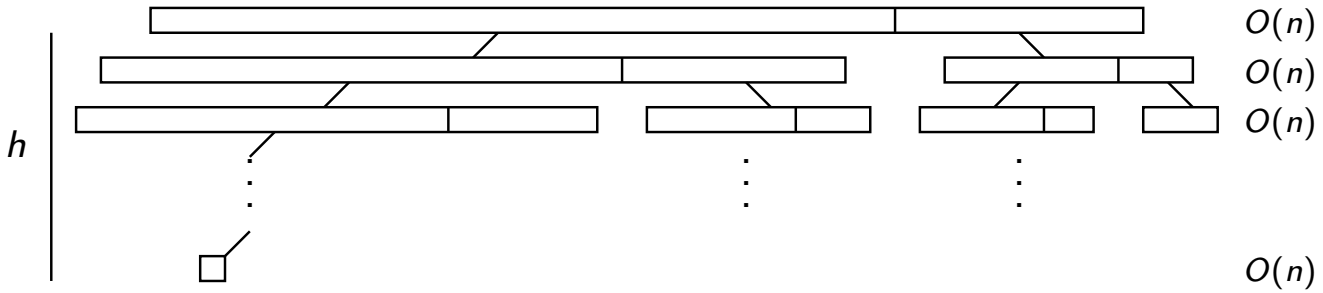
We need to establish the **expected running time**

Sorting: Quicksort

Expected time complexity:

When starting from a random permutation, each recursive call's pivot has a random rank in its list, and therefore is in the middle 50 percent approximately half the time. Hence, in two calls, with high expectation, at least one of the pivots has divided the array in two parts where the largest has size $\leq 3n/4$. $T(n) \approx 2R(n)$

$$R(n) = R(n-k-1) + R(k) + O(n) = R(3n/4) + R(n/4) + O(n)$$



$$\left(\frac{3}{4}\right)^h \cdot n = 1 \quad \Rightarrow \quad h = \log_{4/3} n$$

With $O(n)$ work on each level, expected complexity is $O(n \log n)$

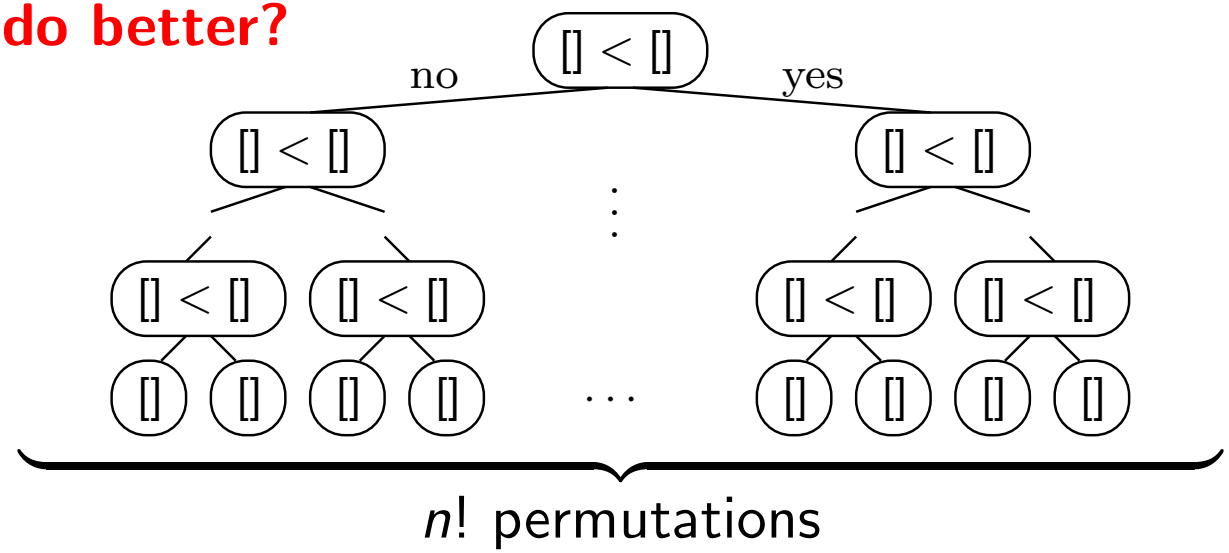
Sorting: A Lower Bound

So, running time for all these sorting methods is:

$$T(n) = T(\alpha \cdot n) + T((1-\alpha) \cdot n) + O(n) = O(n \log n)$$

$\alpha = 1/2,\ 3/4, \ldots$

Can we do better?



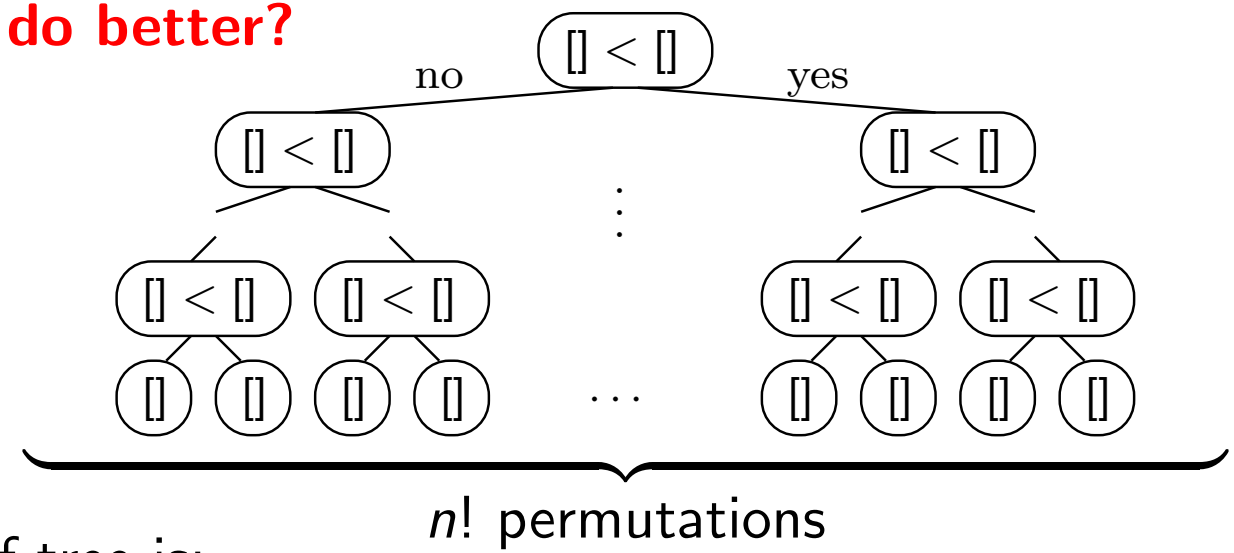
Sorting: A Lower Bound

So, running time for all these sorting methods is:

$$T(n) = T(\alpha \cdot n) + T((1-\alpha) \cdot n) + O(n) = O(n \log n)$$

$\alpha = 1/2,\ 3/4, \ldots$

Can we do better?



Height of tree is:

$$\log n! = \log n(n-1)\cdots 2 \cdot 1 \geq \log n(n-1)\cdots n/2$$

$$\geq \log(n/2)^{n/2} \ = \ n/2 \cdot \log n/2 \ = \ n/2 \cdot \log n - n/2 \ \in \ \Omega(n \log n)$$

Optimal! If we use comparisons to sort!

Divide and Conquer Algorithms

Algorithm design paradigm that:

1. subdivides a problem (if possible) into one or more subproblems (of the same type) having smaller size, [divide]
2. combines the results of the subproblems into a result for the original problem [conquer]

Divide and Conquer Algorithms

Generic divide and conquer algorithm

Algorithm `genericDC`
Input: some X
if $|X|$ is sufficiently small **then**
  return the solution by some brute force method
else
  $X_1, \ldots, X_a \leftarrow$ `subdivide`(X)
  for $i \leftarrow 1, \ldots, a$ **do** $Y_i \leftarrow$ `genericDC`(X_i) **endfor**
  return `combine`$(Y_1, \ldots, Y_a)$
endif
End `genericDC`

Divide and Conquer Algorithms

Generic divide and conquer algorithm

Algorithm `genericDC`
Input: some X
if $|X|$ is sufficiently small **then**
  return the solution by some brute force method
else
  $X_1, \ldots, X_a \leftarrow$ `subdivide`(X)
  for $i \leftarrow 1, \ldots, a$ **do** $Y_i \leftarrow$ `genericDC`(X_i) **endfor**
  return `combine`$(Y_1, \ldots, Y_a)$
endif
End `genericDC`

Running time: $n = |X|$, $n_1 = |X_1|, \ldots, n_l = |X_a|$

$$T(n) = O\big(d(n)\big) + T(n_1) + \cdots + T(n_a) + O\big(c(n)\big)$$

Divide and Conquer Algorithms

If we can subdivide so that each $n_i \leq |X|/b = n/b$, $1 \leq i \leq a$, for constant $b > 1$, then recurrence becomes even simpler

$$T(n) = a \cdot T(n/b) + O\big(d(n) + c(n)\big)$$

can be solved with Master theorem if $d(n) + c(n)$ is polynomial

Thank you for your attention.

Questions?

Comments?