Backtracking with Recursion
Sorting
Finding a data item

• Suppose you have an array (or ArrayList) of \( N \) items. How do you determine if the array contains a particular item?

• Does the form of the array matter?
  • Unsorted
  • Sorted
  • Heap

• What is the complexity of finding an item?
Binary Search

• Search for an integer (22) in an ordered list

\[
\begin{align*}
&\text{0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15} \\
&\hline
&\text{2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37}
\end{align*}
\]

• \( \text{mid} = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor = \left\lfloor \frac{0 + 15}{2} \right\rfloor = 7 \)

  □ \text{target==data[mid], found}
  □ \text{target>data[mid], recur on second half}
  □ \text{target<data[mid], recur on first half}
target = 22

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37

low  mid  high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37

low  mid  high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37

low = mid = high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
View the data as a binary tree

“Binary Search Tree”

Is this a heap?
Can a heap be a Binary Search Tree?
/**
 * The public facing call to array search
 * The array to be searched is a private instance variable
 * @param target the value being searched for
 * @return true if the value is in known, false otherwise
 */
public boolean contains(int target) {
    if (data==null)
        return 0;
    return iSearch(target, 0, data.length-1, 0);
}
Binary Search Code

/**
 * Binary search, recursively on sorted internal array of ints
 * @param target the item to be found
 * @param lo the bottom of the range being searched
 * @param hi the top of the range being searched
 * @param steps the number of steps the search has taken
 * @return true if the target was found
 */
private boolean iSearch(int target, int lo, int hi, int steps) {
    if (lo>hi) return false;
    int mid = (lo+hi)/2;
    System.out.println(target + " " + data[mid] + " " + lo + " " + hi + " " + steps);
    if (data[mid]==target) return true;
    if (data[mid]<target)
        return iSearch(target, mid+1, hi, steps+1);
    else
        return iSearch(target, lo, mid-1, steps+1);
}
Binary Search Analysis

• Each recursive call divides the array in half

• If the array is of size $n$, it divides (and searches) at most $\log_2 n$ times before the current half is of size 1

• $O(\log_2 n)$
Reimplement Binary search with iteration

What parameters does the iterative method need? Does a separate private method even make sense?
Backtracking with Recursion

• Previous examples all progressed linearly to success/failure
• So consider doing binary like search on an unsorted array
  • Need to backtrack and try other directions on failure.
  • Backtracking is when recursion really shines
Recursion and Backtracking

• All examples progress steadily towards an answer.

• Consider a maze. Sometimes you need to backtrack.
  • Will work with maze code in lab.

• Problem: given an english word can you remove one letter and still have an english word.
  • Can you do this repeatedly until only a 2 letter word remains?
  • Consider the word “cored” .. core, ore, or!!!
  • Lets suppose:
    • boolean isInEnglish(String s)
      • return true iff s is an English word
    • String removeNchar(int n, String s)
      • removes the nth character of the string. So removeNchar(0, “dour”) is “our”
Base Cases for Word reducer
public class Worder {
    HashMap<String, Integer> words;
    public Worder() {
        words = new HashMap<>();
        try (BufferedReader br = new BufferedReader(new FileReader("/usr/share/dict/words"))) {
            String l;
            while (null != (l=br.readLine())) {
                words.put(l.trim(), 1);
            }
        } catch (Exception ee) { ee.printStackTrace(); }
    }
    public boolean isEnglish(String s) {
        return words.containsKey(s);
    }
    String removeNchar(int n, String s) {
        if (n>0 && n<s.length()-1) {
            return s.substring(0,n)+s.substring(n+1);
        }
        if (n==0)
            return s.substring(1);
        return s.substring(0, s.length()-1);
    }
    public boolean reducable(String s) {
        System.out.println(s);
        if (s.length()<=2) { return isEnglish(s); }
        if (!isEnglish(s)) return false;
        for (int i=0; i<s.length(); i++) {
            if (reducible(removeNchar(i, s)))
                return true;
        }
        return false;
    }
}
Sorting

• public void sort(Comparable[] arra)
  • change the order of the items in arra
  • All examples will use integers but same statements apply to any Comparable object
  • ideally, do this “in place”.
    • That is do not use any extra memory

• First 3 sort techniques we have already discussed
Selection Sort

- Selection-sort:
  - select the min/max and swap with 0
- priority queue is implemented with an unsorted sequence
- Time:
  - Add: $O(n)$
  - Remove: $O(n^2)$
Example

<table>
<thead>
<tr>
<th>Phase 1 — Inserting</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>7</td>
<td>(7)</td>
</tr>
<tr>
<td>(b)</td>
<td>4</td>
<td>(7,4)</td>
</tr>
<tr>
<td>(g)</td>
<td>()</td>
<td>(7,4,8,2,5,3,9)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phase 2 — Polling</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(2)</td>
<td>(7,4,8,5,3,9)</td>
</tr>
<tr>
<td>(b)</td>
<td>(2,3)</td>
<td>(7,4,8,5,9)</td>
</tr>
<tr>
<td>(c)</td>
<td>(2,3,4)</td>
<td>(7,8,5,9)</td>
</tr>
<tr>
<td>(d)</td>
<td>(2,3,4,5)</td>
<td>(7,8,9)</td>
</tr>
<tr>
<td>(e)</td>
<td>(2,3,4,5,7)</td>
<td>(8,9)</td>
</tr>
<tr>
<td>(f)</td>
<td>(2,3,4,5,7,8)</td>
<td>(9)</td>
</tr>
<tr>
<td>(g)</td>
<td>(2,3,4,5,7,8,9)</td>
<td>()</td>
</tr>
</tbody>
</table>
Insertion Sort

- Insertion-sort:
  - insert/swap the element into the correct sorted position
- Priority queue where the priority queue is implemented with a sorted sequence
- Time:
  - Add: $O(n^2)$
  - Remove: $O(n)$
Example

Phase 1 — Inserting
(a) 7 (7)
(b) 4 (4,7)
(c) 8 (4,7,8)
(d) 2 (2,4,7,8)
(e) 5 (2,4,5,7,8)
(f) 3 (2,3,4,5,7,8)
(g) 9 (2,3,4,5,7,8,9)

Phase 2 — polling
(a) (2) (3,4,5,7,8,9)
(b) (2,3) (4,5,7,8,9)
.. .. ..
(g) (2,3,4,5,7,8,9) ()
Heap Sort

• Heap-sort:
  □ Insertion — no more than \( \log_2(n) \) steps per insertion
  □ Deletion — no more than \( \log_2(n) \) steps per deletion

• priority queue is implemented with a heap

• Time:
  • Add: \( O(n \times \log_2(n)) \) — under some assumptions \( O(n) \).
  • Remove: \( O(n \times \log_2(n)) \)

• Note: with a lot of work can do this without an additional array.
Example

Phase 1 — Inserting
(a) 7  (7)
(b) 4  (4,7)
(c) 8  (4,7,8)
(d) 2  (2,4,8,7)
(e) 5  (2,4,8,7,5)
(f) 3  (2,4,3,7,5,8)
(g) 9  (2,4,3,7,5,8,9)

Phase 2 — polling
(a) (2)  (3,4,7,5,8,9)
(b) (2,3)  (4,5,7,9,8)
..  ..  ..
(g) (2,3,4,5,7,8,9)  ()
## Timing

<table>
<thead>
<tr>
<th>size</th>
<th>selection</th>
<th>Insertion</th>
<th>Insertion</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>2000</td>
<td>8</td>
<td>12</td>
<td>26</td>
<td>3</td>
</tr>
<tr>
<td>4000</td>
<td>24</td>
<td>23</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>8000</td>
<td>96</td>
<td>95</td>
<td>81</td>
<td>10</td>
</tr>
<tr>
<td>16000</td>
<td>370</td>
<td>378</td>
<td>315</td>
<td>17</td>
</tr>
<tr>
<td>32000</td>
<td>1585</td>
<td>1359</td>
<td>1218</td>
<td>36</td>
</tr>
<tr>
<td>64000</td>
<td>5771</td>
<td>5590</td>
<td>4605</td>
<td>77</td>
</tr>
<tr>
<td>128000</td>
<td>23087</td>
<td>21547</td>
<td>19849</td>
<td>161</td>
</tr>
<tr>
<td>256000</td>
<td></td>
<td></td>
<td></td>
<td>345</td>
</tr>
<tr>
<td>512000</td>
<td></td>
<td></td>
<td></td>
<td>1128</td>
</tr>
<tr>
<td>1024000</td>
<td></td>
<td></td>
<td></td>
<td>1973</td>
</tr>
<tr>
<td>2048000</td>
<td></td>
<td></td>
<td></td>
<td>3225</td>
</tr>
<tr>
<td>4096000</td>
<td></td>
<td></td>
<td></td>
<td>7577</td>
</tr>
<tr>
<td>8192000</td>
<td></td>
<td></td>
<td></td>
<td>18586</td>
</tr>
</tbody>
</table>
Batch and Flow

Another consideration is how you get the data and when you produce the sort.

- **BATCH**
  - You get the data all at once and have to produce a sorted list at the time (or later)
- **FLOW** — the data come in over the course of time. At any time, you can be asked to produce a sort list of the data you already have.

Which sorts algorithms perform well / poorly in each situation

- Suppose receive M requests for sort and assume that new data — of size N — arrives unpredictably with respect to M.
  - Selection sort: insert = $O(n)$, retrieve = $O(M \times N^2)$
  - Insertion Sort: insert = $O(N \times N^2)$, retrieve = $O(M)$
  - Heap Sort: insert = $O(N \times \log N)$, retrieve = $O(M \times N \times \log N)$
Divide-and-Conquer

- Divide – the problem (input) into smaller pieces
- Conquer – solve each piece individually, usually recursively
- Combine – the piecewise solutions into a global solution (if needed)
- Usually involves recursion
Merge Sort

• Sort a sequence of numbers $A$, $|A| = n$
• Base case: $|A| = 1$, then it’s already sorted
• General
  □ divide: split $A$ into two halves, each of size $\frac{n}{2}$ ($\left\lfloor \frac{n}{2} \right\rfloor$ and $\left\lceil \frac{n}{2} \right\rceil$)
  □ conquer: sort each half (by calling mergeSort recursively)
  □ combine: merge the two sorted halves into a single sorted list
Example

Input

Output

split

merge

merge

merge
Algorithm

mergeSort(S):
    if S.size() <= 1 return
else
    s1 = S[0,n/2]
    s2 = S[n/2+1,n-1]
    mergeSort(s1)
    mergeSort(s2)
    S = merge(s1, s2)
Merge Algorithm

- The key is the merging process
- How does one merge two sorted lists?
- Each element in $A \cup B$ is considered once
- $O(n)$

Algorithm merge(A, B)
Input sorted A and B
Output sorted A $\cup$ B
S = empty sequence
while (!A.isEmpty() and !B.isEmpty())
  if A.first() < B.first()
    S.addLast(A.removeFirst())
  else
    S.addLast(B.removeFirst())
while (!A.isEmpty())
  S.addLast(A.removeFirst())
while (!B.isEmpty())
  S.addLast(B.removeFirst())
return S
private int[] domerge(int[] list1, int[] list2) {
    int[] rtn = new int[list1.length + list2.length];
    int locr=0, loc1=0, loc2=0;
    while (loc1<list1.length && loc2<list2.length) {
        if (list1[loc1] < list2[loc2]) {
            rtn[locr++]=list2[loc2++];
        } else {
            rtn[locr++]=list1[loc1++];
        }
    }
    for (int i=loc1; i<list1.length; i++)
        rtn[locr++]=list1[i];
    for (int i=loc2; i<list2.length; i++)
        rtn[locr++]=list2[i];
    return rtn;
}
public int[] mergesort(int[] list) {
    return doMergeSort(list, 0, list.length-1);
}

private int[] doMergeSort(int[] list, int strt, int eend) {
    if (eend==strt) {
        int[] tmp = new int[1];
        tmp[0]=list[strt];
        return tmp;
    }
    if (eend<strt) return new int[0];
    int mid = (strt+eend)/2;
    return domerge(mergesort(list, strt, mid), mergesort(list, mid+1, eend));
}
## Timing

<table>
<thead>
<tr>
<th>size</th>
<th>selection</th>
<th>Insertion</th>
<th>Insertion</th>
<th>Heap</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2000</td>
<td>8</td>
<td>12</td>
<td>26</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4000</td>
<td>24</td>
<td>23</td>
<td>20</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>8000</td>
<td>96</td>
<td>95</td>
<td>81</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>16000</td>
<td>370</td>
<td>378</td>
<td>315</td>
<td>17</td>
<td>27</td>
</tr>
<tr>
<td>32000</td>
<td>1585</td>
<td>1359</td>
<td>1218</td>
<td>36</td>
<td>58</td>
</tr>
<tr>
<td>64000</td>
<td>5771</td>
<td>5590</td>
<td>4605</td>
<td>77</td>
<td>119</td>
</tr>
<tr>
<td>128000</td>
<td>23087</td>
<td>21547</td>
<td>19849</td>
<td>161</td>
<td>219</td>
</tr>
<tr>
<td>256000</td>
<td></td>
<td></td>
<td></td>
<td>345</td>
<td>372</td>
</tr>
<tr>
<td>512000</td>
<td></td>
<td></td>
<td></td>
<td>1128</td>
<td>776</td>
</tr>
<tr>
<td>1024000</td>
<td></td>
<td></td>
<td></td>
<td>1973</td>
<td>1631</td>
</tr>
<tr>
<td>2048000</td>
<td></td>
<td></td>
<td></td>
<td>3225</td>
<td>3822</td>
</tr>
<tr>
<td>4096000</td>
<td></td>
<td></td>
<td></td>
<td>7577</td>
<td>6772</td>
</tr>
<tr>
<td>8192000</td>
<td></td>
<td></td>
<td></td>
<td>18586</td>
<td>14159</td>
</tr>
</tbody>
</table>
In-place Merge

- Making new lists is slow!
- How does one merge two sorted lists $A[p, \ldots, m]$ and $A[m+1, \ldots, r]$?
- Use a temp array $B$ and maintain two indices $i$ and $j$, one for each subarray.
MergeSort using one temp array

```java
private int[] array;
private int[] tempMergArr;
private int length;
public int[] mergesort3(int inputArr[]) {
    this.array = inputArr;
    this.length = inputArr.length;
    this.tempMergArr = new int[length];
    doMergeSort3(0, length - 1);
    return array;
}

private void doMergeSort3(int lowerIndex, int higherIndex) {
    if (lowerIndex < higherIndex) {
        int middle = lowerIndex + (higherIndex - lowerIndex) / 2;
        // Below step sorts the left side of the array
        doMergeSort3(lowerIndex, middle);
        // Below step sorts the right side of the array
        doMergeSort3(middle + 1, higherIndex);
        // Now merge both sides
        mergeParts3(lowerIndex, middle, higherIndex);
    }
}
```
private void mergeParts3(int lowerIndex, int middle, int higherIndex) {

    for (int i = lowerIndex; i <= higherIndex; i++) {
        tempMergArr[i] = array[i];
    }

    int i = lowerIndex;
    int j = middle + 1;
    int k = lowerIndex;

    while (i <= middle && j <= higherIndex) {
        if (tempMergArr[i] <= tempMergArr[j]) {
            array[k] = tempMergArr[i];
            i++;
        } else {
            array[k] = tempMergArr[j];
            j++;
        }
        k++;
    }

    while (i <= middle) {
        array[k] = tempMergArr[i];
        k++;
        i++;
    }
}
## Timing

<table>
<thead>
<tr>
<th>size</th>
<th>selection</th>
<th>Insertion</th>
<th>Insertion</th>
<th>Heap</th>
<th>merge</th>
<th>merge (improved)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2000</td>
<td>8</td>
<td>12</td>
<td>26</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4000</td>
<td>24</td>
<td>23</td>
<td>20</td>
<td>5</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8000</td>
<td>96</td>
<td>95</td>
<td>81</td>
<td>10</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>16000</td>
<td>370</td>
<td>378</td>
<td>315</td>
<td>17</td>
<td>27</td>
<td>16</td>
</tr>
<tr>
<td>32000</td>
<td>1585</td>
<td>1359</td>
<td>1218</td>
<td>36</td>
<td>58</td>
<td>32</td>
</tr>
<tr>
<td>64000</td>
<td>5771</td>
<td>5590</td>
<td>4605</td>
<td>77</td>
<td>119</td>
<td>69</td>
</tr>
<tr>
<td>128000</td>
<td>23087</td>
<td>21547</td>
<td>19849</td>
<td>161</td>
<td>219</td>
<td>143</td>
</tr>
<tr>
<td>256000</td>
<td></td>
<td></td>
<td></td>
<td>345</td>
<td>372</td>
<td>294</td>
</tr>
<tr>
<td>512000</td>
<td></td>
<td></td>
<td></td>
<td>1128</td>
<td>776</td>
<td>563</td>
</tr>
<tr>
<td>1024000</td>
<td></td>
<td></td>
<td></td>
<td>1973</td>
<td>1631</td>
<td>1191</td>
</tr>
<tr>
<td>2048000</td>
<td></td>
<td></td>
<td></td>
<td>3225</td>
<td>3822</td>
<td>2412</td>
</tr>
<tr>
<td>4096000</td>
<td></td>
<td></td>
<td></td>
<td>7577</td>
<td>6772</td>
<td>5191</td>
</tr>
<tr>
<td>8192000</td>
<td></td>
<td></td>
<td></td>
<td>18586</td>
<td>14159</td>
<td>10282</td>
</tr>
</tbody>
</table>
# Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>▪ slow</td>
<td>▪ in-place ▪ for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>▪ slow</td>
<td>▪ in-place ▪ for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>▪ fast</td>
<td>▪ in-place ▪ for large data sets (1K — 1M)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>▪ fast</td>
<td>▪ sequential data access ▪ for huge data sets (&gt; 1M)</td>
</tr>
</tbody>
</table>
Mini Homework

14, 6, 18, 2, 13, 7, 8, 9, 3, 17, 5, 10, 11, 12, 15, 19, 16, 0, 1, 4

For the data above, show all the steps of a merge sort, along the lines of slide 12.
Quick Sort

• Another divide-and-conquer sort
  ▫ divide: pick a random element $x$ (pivot) and partition into
    ◆ $L$: $< x$
    ◆ $E$: $= x$
    ◆ $G$: $> x$
  ▫ conquer: sort $L$ and $G$
  ▫ combine: join $L$, $E$ and $G$
Pseudo Code

quickSort(S):
    if (S.size()<2) return
    p = S.last() // first as pivot
    L = E = G = new list()
    partition(S, p)
    quickSort(L)
    quickSort(G)
    S = L+E+G
QS in action

Partition on 6
{2, 4, 3, 1} 6 {7, 7, 9}

Partition on 1
{} 1 {2, 4, 3}

Partition on 3
{2} 3 {4}

Partition on 9
{7, 7} 9 {}

Partition on 7
{7} 7 {}

Assemble
{2, 3, 4} {7, 7}

{1, 2, 3, 4} {7, 7, 9}

{1, 2, 3, 4, 6, 7, 7, 9}

Divide
Conquer
MergeSort, Quicksort, etc

- Quicksort does work on way down in recursion
  - Mergesort does work on way up
- Insertion sort does work on way down
  - Selection sort on way up
- Which one in faster Quick or Merge?
Worst-case Running Time

- When the pivot is the min or max
  - one of $L$ or $G$ has size $n - 1$
  - $T(n) = n + (n - 1) + \ldots + 2 + 1 = O(n^2)$
In-place Quick Sort

- instead of three lists partition rearranges the input list
  - $L: [0, l - 1]$  
  - $E: [l, r]$  
  - $G: [r + 1, n - 1]$  
- Recursive calls on $[0, l - 1]$ and $[r + 1, n - 1]$
public int partition(int arr[], int begin, int end) {
    int pivot = arr[end];
    int insertLoc = (begin-1);
    for (int j = begin; j < end; j++) {
        if (arr[j] <= pivot) {
            insertLoc++;
            int swapTemp = arr[insertLoc];
            arr[insertLoc] = arr[j];
            arr[j] = swapTemp;
        }
    }
    int swapTemp = arr[insertLoc+1];
    arr[insertLoc+1] = arr[end];
    arr[end] = swapTemp;
    return insertLoc+1;
}
private void quickSort(int arr[], int begin, int end) {
    if (begin < end) {
        int partitionIndex = partition(arr, begin, end);
        quickSort(arr, begin, partitionIndex-1);
        quickSort(arr, partitionIndex+1, end);
    }
}
# Speed

**Table 1**

<table>
<thead>
<tr>
<th>size</th>
<th>Insertion</th>
<th>Heap</th>
<th>merge (improved)</th>
<th>Quick</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>11</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2000</td>
<td>26</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4000</td>
<td>20</td>
<td>5</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8000</td>
<td>81</td>
<td>10</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>16000</td>
<td>315</td>
<td>17</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>32000</td>
<td>1218</td>
<td>36</td>
<td>32</td>
<td>30</td>
</tr>
<tr>
<td>64000</td>
<td>4605</td>
<td>77</td>
<td>69</td>
<td>59</td>
</tr>
<tr>
<td>128000</td>
<td>19849</td>
<td>161</td>
<td>143</td>
<td>108</td>
</tr>
<tr>
<td>256000</td>
<td>345</td>
<td>294</td>
<td>219</td>
<td></td>
</tr>
<tr>
<td>512000</td>
<td>1128</td>
<td>563</td>
<td>464</td>
<td></td>
</tr>
<tr>
<td>1024000</td>
<td>1973</td>
<td>1191</td>
<td>955</td>
<td></td>
</tr>
<tr>
<td>2048000</td>
<td>3225</td>
<td>2412</td>
<td>1989</td>
<td></td>
</tr>
<tr>
<td>4096000</td>
<td>7577</td>
<td>5191</td>
<td>4148</td>
<td></td>
</tr>
<tr>
<td>8192000</td>
<td>18586</td>
<td>10282</td>
<td>10101</td>
<td></td>
</tr>
<tr>
<td>16384000</td>
<td></td>
<td></td>
<td></td>
<td>17614</td>
</tr>
<tr>
<td>32768000</td>
<td></td>
<td></td>
<td></td>
<td>37291</td>
</tr>
</tbody>
</table>
Quick and Merge

- Quicksort is reliably quicker than merge
- Quicksort does not need extra memory for auxiliary array
Mini Homework

14, 6, 18, 2, 13, 7, 8, 9, 3, 17, 5, 10, 11, 12, 15, 19, 16, 0, 1, 4

For the data above, show all the steps of a quick sort, following the pattern of slide 19. Always choose the last element as the partitioning element.