CSE 373

MAY 24TH – ANALYSIS AND NON-COMPARISON SORTING
ASSORTED MINUTIAE

• HW6 Out – Due next Wednesday
ASSORTED MINUTIAE

• HW6 Out – Due next Wednesday
  • Only two late days allowed
ASSORTED MINUTIAE

• HW6 Out – Due next Wednesday
  • Only two late days allowed
  • All HW in by Friday, June 2\textsuperscript{nd}
ASSORTED MINUTIAE

- HW6 Out – Due next Wednesday
  - Only two late days allowed
  - All HW in by Friday, June 2\textsuperscript{nd}
  - Regrades also in by that point.
TODAY

• Merge sort and Quick sort examples
TODAY

• Merge sort and Quick sort examples
• Proving $\Omega(n \log n)$ for comparison sorts
TODAY

• Merge sort and Quick sort examples
• Proving $\Omega(n \log n)$ for comparison sorts
• Basics of the Recurrence
TODAY

• Merge sort and Quick sort examples
• Proving $\Omega(n \log n)$ for comparison sorts
• Basics of the Recurrence
• Non-comparison sorting
TODAY

• Merge sort and Quick sort examples
• Proving $\Omega(n \log n)$ for comparison sorts
• Basics of the Recurrence
• Non-comparison sorting
• JUnit
JUnit: Testing Framework

A Java library for unit testing, comes included with Eclipse

- OR can be downloaded for free from the JUnit web site at http://junit.org
- JUnit is distributed as a "JAR" which is a compressed archive containing Java .class files

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class name {
    ...

    @Test
    public void name() { // a test case method
        ...
    }
}
```

A method with @Test is flagged as a JUnit test case and run
JUNIT ASSERTS AND EXCEPTIONS

A test will pass if the assert statements all pass and if no exception thrown. Examples of assert statements:

- `assertTrue(value)`
- `assertFalse(value)`
- `assertEquals(expected, actual)`
- `assertNull(value)`
- `assertNotNull(value)`
- `fail()`

Tests can expect exceptions or timeouts

```java
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```
MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array

First half after sort:

2 4 7 8

Second half after sort:

1 3 5 6

Result:
Merge Example

Merge operation: Use 3 pointers and 1 more array

First half after sort:

2 4 7 8

Second half after sort:

1 3 5 6

Result:
**MERGE EXAMPLE**

**Merge operation:** Use 3 pointers and 1 more array

First half after sort:

\[
\begin{array}{cccc}
2 & 4 & 7 & 8 \\
\end{array}
\]

Second half after sort:

\[
\begin{array}{cccc}
1 & 3 & 5 & 6 \\
\end{array}
\]

Result:

\[
\begin{array}{cccc}
1 & 2 \\
\end{array}
\]
Merge operation: Use 3 pointers and 1 more array

First half after sort:

2 4 7 8

Second half after sort:

1 3 5 6

Result:

1 2 3
MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array

First half after sort:

Second half after sort:

Result:
**MERGE EXAMPLE**

**Merge operation:** Use 3 pointers and 1 more array

First half after sort:

```
| 2 | 4 | 7 | 8 |
```

Second half after sort:

```
| 1 | 3 | 5 | 6 |
```

Result:

```
| 1 | 2 | 3 | 4 | 5 |
```
**MERGE EXAMPLE**

**Merge operation:** Use 3 pointers and 1 more array

First half after sort:

2 4 7 8

Second half after sort:

1 3 5 6

Result:

1 2 3 4 5 6
Merge operation: Use 3 pointers and 1 more array

First half after sort:

\[ \begin{array}{cccc}
2 & 4 & 7 & 8 \\
\end{array} \]  

Second half after sort:

\[ \begin{array}{cccc}
1 & 3 & 5 & 6 \\
\end{array} \]  

Result:

\[ \begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]
**MERGE EXAMPLE**

**Merge operation:** Use 3 pointers and 1 more array

First half after sort:

\[
\begin{array}{cccc}
2 & 4 & 7 & 8 \\
\end{array}
\]

Second half after sort:

\[
\begin{array}{cccc}
1 & 3 & 5 & 6 \\
\end{array}
\]

Result:

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

**After Merge:** copy result into original unsorted array. Or alternate merging between two size n arrays.
QUICK SORT EXAMPLE: DIVIDE

Pivot rule: pick the element at index 0
QUICK SORT EXAMPLE: COMBINE

Combine: this is the order of the elements we’ll care about when combining.

7 3 8 4 5 2 1 6

3 4 5 2 1 6

7 8

2 1

3

4 5 6

1 2

4

5 6

5 6
QUICK SORT EXAMPLE: COMBINE

Combine: put left partition < pivot < right partition

1 2 3 4 5 6 7 8

1 2 3 4 5 6

1 2
1 2
4 5 6
5 6
5 6
MEDIAN PIVOT EXAMPLE

Pick the median of first, middle, and last

\[
\begin{array}{cccccccc}
7 & 2 & 8 & 4 & 5 & 3 & 1 & 6 \\
\end{array}
\]

Median = 6

Swap the median with the first value

\[
\begin{array}{cccccccc}
7 & 2 & 8 & 4 & 5 & 3 & 1 & 6 \\
\end{array}
\]

Pivot is now at index 0, and we’re ready to go

\[
\begin{array}{cccccccc}
6 & 2 & 8 & 4 & 5 & 3 & 1 & 7 \\
\end{array}
\]
PARTITIONING

Conceptually simple, but hardest part to code up correctly

- After picking pivot, need to partition in linear time in place

One approach (there are slightly fancier ones):

1. Put pivot in index $lo$
2. Use two pointers $i$ and $j$, starting at $lo+1$ and $hi-1$
3. while ($i < j$)
   if ($arr[j] > pivot$) $j--$
   else if ($arr[i] < pivot$) $i++$
   else swap $arr[i]$ with $arr[j]$
4. Swap pivot with $arr[i]$  *

*skip step 4 if pivot ends up being least element
EXAMPLE

Step one: pick pivot as median of 3

- $lo = 0$, $hi = 10$

- Step two: move pivot to the $lo$ position
QUICK SORT PARTITION EXAMPLE
ASYMPTOTIC RUNTIME OF RECURSION

Recurrence Definition:
Recurrence Definition:

A recurrence is a recursive definition of a function in terms of smaller values.

Example: Fibonacci numbers.
ASYMPTOTIC RUNTIME OF RECURSION

Recurrence Definition:

A recurrence is a recursive definition of a function in terms of smaller values.

Example: Fibonacci numbers.

To analyze the runtime of recursive code, we use a recurrence by splitting the work into two pieces:

- Non-Recursive Work
- Recursive Work
RECURSIVE VERSION OF SUM:

```java
int sum(int[] arr) {
    return help(arr, 0, arr.length);
}

int help(int[] arr, int lo, int hi) {
    if (lo == hi) return 0;
    if (lo == hi - 1) return arr[lo];
    int mid = (hi + lo) / 2;
    return help(arr, lo, mid) + help(arr, mid, hi);
}
```

What’s the recurrence $T(n)$?

- Non-Recursive Work:
- Recursive Work:
RECURSIVE VERSION OF SUM:

```c
int sum(int[] arr) {
    return help(arr, 0, arr.length);
}
int help(int[] arr, int lo, int hi) {
    if(lo==hi) return 0;
    if(lo==hi-1) return arr[lo];
    int mid = (hi+lo)/2;
    return help(arr, lo, mid) + help(arr, mid, hi);
}
```

What’s the recurrence $T(n)$?

- Non-Recursive Work: $O(1)$
- Recursive Work: $T(n/2) \times 2$ halves

$T(n) = O(1) + 2T(n/2)$
SOLVING THAT
RECURRENCE RELATION

1. Determine the recurrence relation. What is the base case?
   • If $T(1) = 1$, then $T(n) = 1 + 2*T(n/2)$

2. “Expand” the original relation to find an equivalent general expression in terms of the number of expansions.
   • $T(n) = 1 + 2 * T(n/2)$
     $= 1 + 2 + 2 * T(n/4)$
     $= 1 + 2 + 4 + \ldots$ for $\log(n)$ times
     $= \ldots$
     $= 2^{(\log n)} - 1$

3. Find a closed-form expression by setting the number of expansions to a value which reduces the problem to a base case
   • So $T(n)$ is $O(n)$

Explanation: it adds each number once while doing little else
1. Determine the recurrence relation. What is the base case?
   • If \( T(n) = 10 + T(n/2) \) and \( T(1) = 10 \)

2. “Expand” the original relation to find an equivalent general expression in terms of the number of expansions.
   • \( T(n) = 10 + 10 + T(n/4) \)
     = \( 10 + 10 + 10 + T(n/8) \)
     = …
     = \( 10k + T(n/(2^k)) \)

3. Find a closed-form expression by setting the number of expansions to a value which reduces the problem to a base case
   • \( n/(2^k) = 1 \) means \( n = 2^k \) means \( k = \log_2 n \)
   • So \( T(n) = 10 \log_2 n + 8 \) (get to base case and do it)
   • So \( T(n) \) is \( O(\log n) \)
REALLY COMMON
RECURRENTENCES

You can recognize some really common recurrences:

\[
T(n) = O(1) + T(n-1) \\
T(n) = O(1) + 2T(n/2) \\
T(n) = O(1) + T(n/2) \\
T(n) = O(1) + 2T(n-1) \\
T(n) = O(n) + T(n-1) \\
T(n) = O(n) + T(n/2) \\
T(n) = O(n) + 2T(n/2)
\]

linear
linear
logarithmic $O(\log n)$
exponential
quadratic
linear
$O(n \log n)$ (divide
and conquer sort)

Note big-Oh can also use more than one variable
Example: can sum all elements of an $n$-by-$m$ matrix in $O(nm)$
**QUICK SORT ANALYSIS**

Best-case: Pivot is always the median

\[ T(0) = T(1) = 1 \]
\[ T(n) = 2T(n/2) + n \quad \text{-- linear-time partition} \]

Same recurrence as mergesort: \( O(n \log n) \)

Worst-case: Pivot is always smallest or largest element

\[ T(0) = T(1) = 1 \]
\[ T(n) = 1T(n-1) + n \]

Basically same recurrence as selection sort: \( O(n^2) \)

Average-case (e.g., with random pivot)

- \( O(n \log n) \), not responsible for proof
HOW FAST CAN WE SORT?

Heapsort & mergesort have $O(n \log n)$ worst-case running time.

Quicksort has $O(n \log n)$ average-case running time.

- **Assuming our comparison model:** The only operation an algorithm can perform on data items is a 2-element comparison. There is no lower asymptotic complexity, such as $O(n)$ or $O(n \log \log n)$. 
COUNTING COMPARISONS

No matter what the algorithm is, it cannot make progress without doing comparisons

• **Intuition**: Each comparison can *at best* eliminate *half* the remaining possibilities of possible orderings

Can represent this process as a *decision tree*

• Nodes contain “set of remaining possibilities”
• Edges are “answers from a comparison”
• The algorithm does not actually build the tree; it’s what our *proof* uses to represent “the most the algorithm could know so far” as the algorithm progresses
COUNTING COMPARISONS

No matter what the algorithm is, it cannot make progress without doing comparisons
COUNTING COMPARISONS

No matter what the algorithm is, it cannot make progress without doing comparisons

- **Intuition**: Each comparison can *at best* eliminate *half* the remaining possibilities of possible orderings
COUNTING COMPARISONS

No matter what the algorithm is, it cannot make progress without doing comparisons

- Intuition: Each comparison can at best eliminate half the remaining possibilities of possible orderings

Can represent this process as a decision tree
COUNTING COMPARISONS

No matter what the algorithm is, it cannot make progress without doing comparisons

- **Intuition**: Each comparison can *at best* eliminate *half* the remaining possibilities of possible orderings.

Can represent this process as a *decision tree*

- Nodes contain “set of remaining possibilities”
- Edges are “answers from a comparison”
- The algorithm does not actually build the tree; it’s what our *proof* uses to represent “the most the algorithm could know so far” as the algorithm progresses.
The leaves contain all the possible orderings of $a$, $b$, $c$. 

The decision tree for $N = 3$ is as follows:

- **Root Node:** $a < b < c, b < c < a, a < c < b, c < a < b, a < b, c < a, b < c, a < c, b < a, c < b$

- **Left Branch:**
  - $a < b$
    - **Left Leaf:** $a < b < c$
    - **Right Leaf:** $a < c < b$
  - $a > c$
    - **Left Leaf:** $a < b < c$
    - **Right Leaf:** $a < c < b$

- **Right Branch:**
  - $a > b$
    - **Left Leaf:** $b < a < c$
    - **Right Leaf:** $b < c < a$
  - **Left Leaf:** $b < a < c$
  - **Right Leaf:** $b < c < a$

- **Left Leaf:** $b < a < c$
  - **Right Leaf:** $b < c < a$
  - **Right Leaf:** $b < a < c$

- **Right Leaf:** $b < a < c$
  - **Right Leaf:** $b < c < a$

- **Right Leaf:** $b < a < c$
  - **Right Leaf:** $b < c < a$

**Conclusion:**

- The leaves contain all the possible orderings of $a$, $b$, $c$. 

**Decision Tree Diagram:***

- Node 1: $a < b < c$, $b < c < a$, $a < c < b$, $c < a < b$
  - $a < b$
    - $a < c$:
      - $a < b < c$
      - $a < c < b$
      - $b < c$:
        - $a < b < c$
        - $a < c < b$
    - $a > c$:
      - $a < b < c$
      - $a < c < b$
  - $a > b$
    - $b < c$:
      - $b < a < c$
      - $b < c < a$
    - $b > c$:
      - $c < a$
      - $c > a$
      - $b < c < a$
      - $b < a < c$
EXAMPLE IF A < C < B

a < b < c, b < c < a,
a < c < b, c < a < b,
b < a < c, c < b < a

possible orders

a < b

a < c

a > c

a > b

b < a < c
b < c < a
c < b < a

b < c

b > c

b < a < c
b < c < a
c < b < a

b > a < c
b < c < a

actual order

a < b < c
a < c < b
DECISION TREE
A binary tree because each comparison has 2 outcomes (we’re comparing 2 elements at a time)
Because any data is possible, any algorithm needs to ask enough questions to produce all orderings.

The facts we can get from that:

1. Each ordering is a different leaf (only one is correct)
2. Running any algorithm on any input will at best correspond to a root-to-leaf path in some decision tree. Worst number of comparisons is the longest path from root-to-leaf in the decision tree for input size n
3. There is no worst-case running time better than the height of a tree with <num possible orderings> leaves
POSSIBLE ORDERINGS

Assume we have \( n \) elements to sort. How many permutations of the elements (possible orderings)?

- For simplicity, assume none are equal (no duplicates)

Example, \( n=3 \)

\[
\begin{align*}
\text{a[0]} &< \text{a[1]} < \text{a[2]} & \quad \text{a[0]} &< \text{a[2]} < \text{a[1]} \\
\text{a[1]} &< \text{a[0]} < \text{a[2]} & \quad \text{a[1]} &< \text{a[2]} < \text{a[0]} \\
\text{a[1]} &< \text{a[2]} < \text{a[0]} & \quad \text{a[2]} &< \text{a[0]} < \text{a[1]} \\
\text{a[2]} &< \text{a[1]} < \text{a[0]} & \quad \text{a[2]} &< \text{a[1]} < \text{a[0]}
\end{align*}
\]

In general, \( n \) choices for least element, \( n-1 \) for next, \( n-2 \) for next, …

- \( n(n-1)(n-2)\ldots(2)(1) = n! \) possible orderings

That means with \( n! \) possible leaves, best height for tree is \( \log(n!) \), given that best case tree splits leaves in half at each branch
RUNTIME

That proves runtime is at least $\Omega(\log (n!))$. Can we write that more clearly?

$$
\lg(n!) = \lg(n(n-1)(n-2)...1) \\
= \lg(n) + \lg(n-1) + ... + \lg\left(\frac{n}{2}\right) + \lg\left(\frac{n}{2} - 1\right) + ... + \lg(1) \\
\geq \lg(n) + \lg(n-1) + ... + \lg\left(\frac{n}{2}\right) \\
\geq \left(\frac{n}{2}\right) \lg\left(\frac{n}{2}\right) \\
= \left(\frac{n}{2}\right) (\lg n - \lg 2) \\
= \frac{n \lg n}{2} - \frac{n}{2} \\
\in \Omega(n \lg(n))
$$

Nice! Any sorting algorithm must do at best $(1/2)^* (n \log n - n)$ comparisons: $\Omega(n \log n)$
SORTING

• This is the lower bound for comparison sorts
SORTING

- This is the lower bound for comparison sorts
- How can non-comparison sorts work better?
SORTING

• This is the lower bound for comparison sorts
• How can non-comparison sorts work better?
  • They need to know something about the data
SORTING

• This is the lower bound for comparison sorts
• How can non-comparison sorts work better?
  • They need to know something about the data
• Strings and Ints are very well ordered
SORTING

• This is the lower bound for comparison sorts
• How can non-comparison sorts work better?
  • They need to know something about the data
• Strings and Ints are very well ordered
  • If I told you to put “Apple” into a list of words, where would you put it?
BUCKETSORT

If all values to be sorted are known to be integers between 1 and $K$ (or any small range):

- Create an array of size $K$
- Put each element in its proper bucket (a.k.a. bin)
- *If* data is only integers, no need to store more than a *count* of how times that bucket has been used

**Output result via linear pass through array of buckets**

<table>
<thead>
<tr>
<th>count array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

- **Example:**
  
  K=5
  
  input (5, 1, 3, 4, 3, 2, 1, 1, 5, 4, 5)
  
  output: 1, 1, 1, 2, 3, 3, 4, 4, 5, 5, 5
ANALYZING BUCKET SORT

Overall: $O(n+K)$
  - Linear in $n$, but also linear in $K$

Good when $K$ is smaller (or not much larger) than $n$
  - We don’t spend time doing comparisons of duplicates

Bad when $K$ is much larger than $n$
  - Wasted space; wasted time during linear $O(K)$ pass

For data in addition to integer keys, use list at each bucket
BUCKET SORT

Most real lists aren’t just keys; we have data
Each bucket is a list (say, linked list)
To add to a bucket, insert in $O(1)$ (at beginning, or keep pointer to last element)

<table>
<thead>
<tr>
<th>count array</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>Rocky V</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>Harry Potter</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>Casablanca</td>
<td>Star Wars</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>Casablanca</td>
<td>Star Wars</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>5: Star Wars Original Trilogy</td>
<td>1: Rocky V</td>
</tr>
<tr>
<td>1: Rocky V</td>
<td>3: Harry Potter</td>
<td>5: Casablanca</td>
<td>5: Star Wars</td>
<td></td>
</tr>
</tbody>
</table>

Example: Movie ratings; scale 1-5
Input:

Result: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars
Easy to keep ‘stable’; Casablanca still before Star Wars
RADIX SORT

Radix = “the base of a number system”

• Examples will use base 10 because we are used to that
• In implementations use larger numbers
  • For example, for ASCII strings, might use 128

Idea:

• Bucket sort on one digit at a time
  • Number of buckets = radix
  • Starting with least significant digit
  • Keeping sort stable
• Do one pass per digit
• Invariant: After $k$ passes (digits), the last $k$ digits are sorted
**RADIX SORT EXAMPLE**

Radix = 10

**Input:** 478, 537, 9, 721, 3, 38, 143, 67

3 passes (input is 3 digits at max), on each pass, stable sort the input highlighted in yellow

```
4 7 8  7 2 1  0 0 3  0 0 3
5 3 7  0 0 3  0 0 9  0 0 9
0 0 9  1 4 3  7 2 1  0 3 8
7 2 1  5 3 7  5 3 7  0 6 7
0 0 3  4 7 8  0 3 8  1 4 3
0 3 8  0 3 8  0 6 7  4 7 8
1 4 3  0 0 9  5 3 7  7 2 1
0 6 7  0 0 9
```
ANALYSIS

Input size: $n$
Number of buckets = Radix: $B$
Number of passes = “Digits”: $P$

Work per pass is 1 bucket sort: $O(B+n)$

Total work is $O(P(B+n))$

Compared to comparison sorts, sometimes a win, but often not

• Example: Strings of English letters up to length 15
  • Run-time proportional to: 15*(52 + $n$)
  • This is less than $n \log n$ only if $n > 33,000$
  • Of course, cross-over point depends on constant factors of the implementations
SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small $n$

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts
SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small $n$

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n \log n)$ sorts

- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
  - Often fastest, but depends on costs of comparisons/copies
SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small $n$

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n \log n)$ sorts

- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
  - Often fastest, but depends on costs of comparisons/copies

$\Omega (n \log n)$ is worst-case and average lower-bound for sorting by comparisons
SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small $n$

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n \log n)$ sorts

- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
  - Often fastest, but depends on costs of comparisons/copies

$\Omega (n \log n)$ is worst-case and average lower-bound for sorting by comparisons

Non-comparison sorts

- Bucket sort good for small number of possible key values
- Radix sort uses fewer buckets and more phases

Best way to sort? It depends!
SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small $n$

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n \log n)$ sorts

- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
  - Often fastest, but depends on costs of comparisons/copies

$\Omega (n \log n)$ is worst-case and average lower-bound for sorting by comparisons

Non-comparison sorts

- Bucket sort good for small number of possible key values
- Radix sort uses fewer buckets and more phases

Best way to sort? It depends!