Announcements

- Programming focus again, start early and make sure you can do the things we cover in class
- See me if something doesn’t click

- Reading:
  - Skim 7.2, 7.4, 7.5, 7.6, 7.7
Outline

- Sorting Algorithms
  - Basics
    - Slow
    - medium
  - Complicated
    - How fast can we go
    - How they work
  - DS to support them
Preview

- In the next few weeks
  - Inheritance
  - Class relationships

- Homework posted:
  - Problem sets (due apr 2)
  - Viruses and Virus checking program
    - Tentative due date: apr 2, will extend if needed
For homework

- Outline of the problem

- What you need to learn in java
  - Reading/writing files
  - In binary form
  - Using hashtables in multiple ways
  - Adopting it for faster processing
  - Saving live data structures for later use

- Will cover practical java examples on all this on Wednesday...
Sort a bunch of items

- So its straightforward to sort in $O(N^2)$ time
  - Insertion sort
  - Selection sort
  - Bubble sort
Selection sort

- 2 arrays, sorted and unsorted
- keep choosing min from the unsorted list and append to sorted
Bubble Sort

- Anyone ??

- iterate and swap out of ordered elements
Insertion sort

- this is the quickest of the $O(N^2)$ algorithms for small sets
Insertion sort algorithm…

- sort 1st element
- sort first 2
- sort first 3
- etc
```java
insertionSort(int arr[]) {
    int i = 1;
    while (i < arr.length) {
        insert(a, i, arr[i]);
        i = i + 1;
    }
}

insert(int a[], int length, value) {
    int i = length - 1;
    while (i ≥ 0 and a[i] > value) {
        a[i + 1] = a[i];
        i = i - 1;
    }
    a[i + 1] = value;
}
```
/**
 * Simple insertion sort.
 * @param a an array of Comparable items.
 */

public static <AnyType extends Comparable<? super AnyType>>
void insertionSort( AnyType [] a )
{
    int j;

    for( int p = 1; p < a.length; p++ )
    {
        AnyType tmp = a[ p ];
        for( j = p; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
implementation

- so would implementation of the underlying list affect the runtime?
  - how?

- any ideas why these are slow??
  - can you prove it?
Lower Bound

- This is an analysis for simple sorts

- Inversion:
  - an ordered pair \((i,j)\) such that \(i < j\) and \(a[i] > a[j]\)

- Can you find the inversions?
- \([45, 34, 23, 35, 59]\)
So if we swap adjacent items, we only solve at most one inversion

this leads to our slowdown

any ideas?
Theory

- before continuing....

- What would be the average number of inversion on an array of N elements ??
Average inversions

\[ \frac{N(N-1)}{4} \]

- Let L be an unsorted list of elements
- Let \( L_r \) be the reverse of that list
- Any two elements are inverted either in L or \( L_r \)

- Need to look at the pairs
\[
\frac{N(N-1)}{2}
\]

- pairs in L
- on average \( \frac{1}{2} \) will be inverted
- so how does swapping affect the number?
so how to do better than $N^2$?
Shell sort

- idea was to look at elements which are not adjacent

- Example:
  - look at every 8th element and do insert sort on those
    - slide window
  - Now look at every 4th
  - Every 2nd

- Increment series
Increment series

- we have an increment series $h_1, h_2, \ldots, h_k$
- $h_k$ must be less than $N$
- $h_1$ must be 1
  - why?
- each step keeps it sorted for last step
An array is $h_k$ sorted

for every $i$ $a[i] \leq a[i + h_k]$

we use diminishing increments

Example
as long as last increment is 1, we are guaranteed to sort

if we only do 1
  what is it?

lets look at the code
void shellsort(int a[], int len) {
    for( int gap = len/2; gap > 0; gap /=2) {
        for(int i=gap; i<len; i++) {
            int tmp = a[i];
            int j=i;
            for(;j>=gap && tmp < aj-gap]; j-=gap) {
                a[j] = a[j-gap];
            }
            a[j] = tmp;
        }
    }
}
/**
 * Shellsort, using Shell's (poor) increments.
 * @param a an array of Comparable items.
 */

public static <AnyType extends Comparable<? super AnyType>>
void shellsort( AnyType [] a )
{
    int j;

    for( int gap = a.length / 2; gap > 0; gap /= 2 )
        for( int i = gap; i < a.length; i++ )
            {
                AnyType tmp = a[ i ];
                for( j = i; j >= gap &&
                     tmp.compareTo( a[ j - gap ] ) < 0; j -= gap )
                    a[ j ] = a[ j - gap ];
                a[ j ] = tmp;
            }
So what is the increment series here??

1 2 4 8 16 .. $2^k$ $\Theta(N^2)$

Hubert
- 1 3 7 .. $2^k$-1 $\Theta(N^{1.5})$

bizarre sequences
- $\Theta(N^{1.3})$
worst case runtime

<table>
<thead>
<tr>
<th>Start</th>
<th>1</th>
<th>9</th>
<th>2</th>
<th>10</th>
<th>3</th>
<th>11</th>
<th>4</th>
<th>12</th>
<th>5</th>
<th>13</th>
<th>6</th>
<th>14</th>
<th>7</th>
<th>15</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 8-sort</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>11</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>6</td>
<td>14</td>
<td>7</td>
<td>15</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>After 4-sort</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>11</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>6</td>
<td>14</td>
<td>7</td>
<td>15</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>After 2-sort</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>11</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>6</td>
<td>14</td>
<td>7</td>
<td>15</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>After 1-sort</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>
Heapsort

- Heap sort worst case $O(N \log N)$
  - average is slightly better
    - $2N(\log N - \log \log N - 4)$

- can save space using the same array
  - example
Better times

- lets start with better than $n^2$ sorting
merge sort

- if list has one element
  - return

- else
  - mergesort left half
  - mergesort right half
  - merge 2 halves

- Example
/**
 * Internal method for heapsort.
 * @param i the index of an item in the heap.
 * @return the index of the left child.
 */
private static int leftChild( int i )
{
    return 2 * i + 1;
}

/**
 * Internal method for heapsort that is used in deleteMax and buildHeap.
 * @param a an array of Comparable items.
 * @param i the position from which to percolate down.
 * @param n the logical size of the binary heap.
 */
private static <AnyType extends Comparable<? super AnyType>>
void percDown( AnyType[] a, int i, int n )
{
    int child;
    AnyType tmp;
    for( tmp = a[ i ]; leftChild( i ) < n; i = child )
    {
        child = leftChild( i );
        if( child != n - 1 &\& a[ child ].compareTo( a[ child + 1 ] ) < 0 )
            child++;
        if( tmp.compareTo( a[ child ] ) < 0 )
            a[ i ] = a[ child ];
        else
            break;
    }
    a[ i ] = tmp;
}

/**
 * Standard heapsort.
 * @param a an array of Comparable items.
 */
public static <AnyType extends Comparable<? super AnyType>>
void heapsort( AnyType[] a )
{
    for( int i = a.length / 2; i >= 0; i-- ) /* buildHeap */
        percDown( a, i, a.length );
    for( int i = a.length - 1; i > 0; i-- )
    {
        swapReferences( a, 0, i ); /* deleteMax */
        percDown( a, 0, i );
    }
/**
 * Internal method that makes recursive calls.
 * @param a an array of Comparable items.
 * @param tmpArray an array to place the merged result.
 * @param left the left-most index of the subarray.
 * @param right the right-most index of the subarray.
 */

private static <AnyType extends Comparable<? super AnyType>>
void mergeSort( AnyType [] a, AnyType [] tmpArray, int left, int right )
{
    if( left < right )
    {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}

/**
 * Mergesort algorithm.
 * @param a an array of Comparable items.
 */

public static <AnyType extends Comparable<? super AnyType>>
void mergeSort( AnyType [] a )
{
    AnyType [] tmpArray = (AnyType[]) new Comparable[ a.length ];
    mergeSort( a, tmpArray, 0, a.length - 1 );
}
Analysis

- Lets do some simple analysis on mergesort running times

- Assume we have N items
  - N being a power of 2 so we can split nicely
  - if N is one, constant time to mergesort
  - else its 2 * N/2 mergesorts
- Define function
- \( T(N) = \) time to mergesort \( N \) items
- \( T(1) = 1 \)
- \( T(N) = 2T(n/2)+N \)
- how to solve this ??
First method: Telescoping

- trick is what to divide by

\[
\frac{T(N)}{N} = \frac{2T\left(\frac{N}{2}\right)}{N} + 1
\]

- what happens when you add 2 consecutive ones ??

\[
\frac{T(N)}{N} = \frac{T\left(\frac{N}{2}\right)}{N} + 1
\]

now \_ for \_ next

\[
\frac{T\left(\frac{N}{2}\right)}{\left(\frac{N}{2}\right)} = \frac{T\left(\frac{N}{4}\right)}{\left(\frac{N}{4}\right)} + 1
\]

... 

\[
\frac{T(2)}{2} = \frac{T(1)}{1} + 1
\]
Solution

\[
\frac{T(N)}{N} = \frac{T(1)}{1} + \log N
\]

\[
T(N) = N \times T(1) + N \log N
\]
limitations

- telescoping is great, but sometimes hard to find what to divide by

- substitution is another method
substitution

- $T(N) = 2T(N/2) + N$

- sub $N/2$

- $T(N/2) = 2T(N/4) + N/2$

- go back to original

- $T(N) = 4T(N/4) + 2N$
what do you get in the end ??
\[ T(N) = 2^k T(N/2^k) + KN \]
bottom line

- telescoping
  - more scratch work
- substitution
  - more brute force
  - easier when don’t have a clue
end of the day

- Mergesort
  - $O(n \log n)$

- if so good why not the default one?
reality

- requires extra temporary array
- copying is slow....sometimes
  - constant time to the big O runtime will catch up to you

- Great for external sorting
Next

- cue dramatic music

- QUICKSORT
Quick sort

- fastest currently known sort
  - Average $N \log N$
  - Worst: $N^2$
Quicksort

- if one element return
- else
  - pick a pivot from the list
  - split the list around the pivot
  - return quicksort(left) + pivot + quicksort(right)

- Lets do an example
issues

- How does worst case happen?
- how to pick the pivot??
Pivot #1

- use the first element of the list

- pro/cons ?
sorted list will always be $N^2$
Pivot #2

- choose random element for pivot

- pro/cons?
- great performance
- expensive to generate random number
Pivot #3

- Choose median value from the list

- pro/cons?
hmmm don’t you need a sorted list to get median?

actually there is a linear algorithm for this 😊 will be doing it on homework
Pivot #4

- Median of 3

- since #3 isn't cheap, can grab 3 elements and take median
  - can even use random if you don’t mind
Next

- Java file manipulations
- Java generics
- Java serializable
- Java comparable