Last Time

- Sort structure and “rules”
- Selection sort
- Insertion sort
- Comparing sorting methods

- **Important note:** Insertion Sort performs very efficiently with partially sorted arrays.
Shell Sort

```c
int h = 1;
while (h < N/3) {
    h = 3*h + 1;
} //while
while (h >= 1) {
    for (int i = h; i < N; i++) {
        for (int j = i;
            j >= h && less(a[j], a[j-h]);
            j -= h) {
            exch(a, j, j-h);
        } //for
    } //for
    h = h/3;
} //while
```
Shell Sort

- Values for $h$: 1, 4, 13, 40, 121, 364, 1093, ...
- In Insertion Sort, if the item with the smallest key is at the right end of the array, you need $N - 1$ exchanges to move it in place.
  - **Solution**: If two array items are far apart and definitely need to be swapped, why not just do one exchange instead?
  - Shell Sort improves over Insertion Sort by creating interleaved partially sorted arrays that can later be efficiently sorted by Insertion Sort.
Shell Sort Visual

<table>
<thead>
<tr>
<th>Input</th>
<th>SHELL SORT EXAMPLE</th>
<th>1-sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>13-sort</td>
<td>SHELL SORT EXAMPLE</td>
<td>E L A M H L E P S O L T S X R</td>
</tr>
<tr>
<td>4-sort</td>
<td>SHELL SORT EXAMPLE</td>
<td>A E E L M H L E P S O L T S X R</td>
</tr>
</tbody>
</table>

01/27/2015 Improving Sort Efficiency
Shell Sort Visual

- Input
- 40-sorted
- 13-sorted
- 4-sorted
- Result
Shell Sort Evaluation

• Sequence of decreasing values $\frac{1}{2} (3^k - 1)$.
  – Starts at the smallest increment $\geq \lceil n/3 \rceil$ and decreasing to 1. This is called an increment sequence. (how to we pick a sequence to use?)

• When an $h$-sorted array is $k$-sorted, it remains $h$-sorted.

• Worst case for Shell Sort is $O(n^{3/2})$.

• Average case... depends on the increment sequence selected.

• Practical improvement depends on array size.
Picking a Sequence

• No provably best sequence has been found.
  – Depends on the number of increments, arithmetical interactions among the increments (common divisors), etc.
  – *(could a best sequence exist?)*

• What we pick should be easy to compute and use.
  – Simple sequences can perform almost as well as more sophisticated sequences.
What Have We Learned?

• The simple increment sequence modification brought our run time down from $O(n^2)$ to $O(n^{3/2})$.
  – Finding things like this is a primary goal for many algorithm design problems.

• Shell Sort pros:
  – An acceptable run time for moderately large arrays. (Very good algorithms may only run twice as fast except for very large arrays.)
  – Doesn’t require much code.
  – No extra space required.
Mergesort

• Basic idea: Split an array into two halves, sort them, and then merge them back into a single sorted array.
  – (How do we sort each half?)
Naïve Merge

......
public void merge(Comparable[] a, int lo, int mid, int hi) {
    int i = lo, j = mid + 1;
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    } //for
    for (int k = lo; k <= hi; k++) {
        if (i > mid) {
            a[k] = aux[j++];
        } else if (j > hi) {
            a[k] = aux[i++];
        } else if (less(aux[j], aux[i])) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        } //if-else
    } //for
} //merge
public void merge(Comparable[] a, int lo, int mid, int hi) {
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    } //for
    for (int k = lo; k <= hi; k++) {
        if (i > mid) {
            a[k] = aux[j++];
        } else if (j > hi) {
            a[k] = aux[i++];
        } else if (less(aux[j], aux[i])) {
            a[k] = aux[j++];
        } else {
            a[k] = aux[i++];
        } //if-else
    } //for
} //merge
Abstract In-Place Merge Visual
Abstract In-Place Merge

• Still uses extra space, but not a substantial amount more.

• We only need to allocate memory for aux once; then we can continually overwrite it as we work through the data.

• The merge function actually handles the sort! Now we just need to structure our merge() calls:
  – Top-down mergesort
  – Bottom-up mergesort
private static Comparable[] aux;

public static void sort(Comparable[] a) {
    aux = new Comparable[a.length];
    sort(a, 0, a.length-1);
} //sort

private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) {
        return;
    } //if
    int mid = lo + (hi - lo) / 2;
    sort(a, lo, mid);
    sort(a, mid+1, hi);
    merge(a, lo, mid, hi);
} //sort
Top-Down Mergesort Visual
Top-Down Mergesort Evaluation

- This is called a **divide-and-conquer** algorithm – it recursively breaks down the problem into 2+ subproblems of the same (or related) type, until they become simple enough to solve directly.

- Think of it as a proof by induction in code form – we can sort a simple array, and then we can sort a complex array from a previous pair of arrays.
Top-Down Mergesort Evaluation

• How many compares does this algorithm perform?
  – Let $C(N)$ be the number of compares needed to sort an array of length $N$.
    • $C(0) = C(1) = 0$
  – Upper bound for the number of compares required:
    • $C(N) \leq C(N/2) + C(N/2) + N$
  – Lower bound for the number of compares required:
    • $C(N) \leq C(N/2) + C(N/2) + N/2$
  – Can we get an exact solution?
Top-Down Mergesort Evaluation

- To make things easy, let’s say that $N$ is a power of 2 (say $N = 2^n$). This tells us that $N/2 = 2^{n-1}$.
  - $C(2^n) = C(2^{n-1}) + C(2^{n-1}) + 2^n$
  - $C(2^n) = 2C(2^{n-1}) + 2^n$
  - $C(2^n)/2^n = 2C(2^{n-1})/2^n + 2^n/2^n$
  - $C(2^n)/2^n = C(2^{n-1})/2^{n-1} + 1$
  - $C(2^n)/2^n = C(2^{n-2})/2^{n-2} + 1 + 1$
  - ......
  - $C(2^n)/2^n = C(2^0)/2^0 + n$
  - $C(2^n)/2^n = 0/1 + n$
  - $C(2^n)/2^n = n$
  - $C(2^n) = n \times 2^n$
  - $C(2^n) = C(N) = n \times 2^n = N \times \log(N)$

Divide both sides by $2^n$

Apply same equation to $2^{n-1}$

Repeat n-1 times

Multiply by $2^n$

Convert back from $n$ to $N$
Top-Down Mergesort Evaluation

• How many compares does this algorithm perform?
  – $N \times \log(N)$

• How many exchanges does this algorithm perform?
  – Well... we don’t actually do exchanges. Instead, we do array accesses.
  – Each merge uses $2N$ for the copy, $2N$ for the move back, and at most $2N$ for compares.
  – $6N \times \log(N)$
Top-Down Mergesort Improvements

• Now we can sort in $N \times \log(N)$ time – a substantial improvement over the $N^2$ time of Insertion and Selection Sorts.

• We know that Insertion Sort is efficient for small arrays, so if we switch to Insertion Sort once the problem is broken down below some threshold, we can improve Mergesort by 10-15%.

• We can reduce the run time to be linear for arrays that are already sorted, by adding a test to skip the call to $\text{merge}()$ if $a[mid] \leq a[mid+1]$. 
Top-Down Mergesort Improvements
Bottom-Up Mergesort

private static Comparable[] aux;

public static void sort(Comparable[] a) {
    int N = a.length;
    aux = new Comparable[N];
    for (int sz = 1; sz < N; sz = sz+sz) {
        for (int lo = 0; lo < N-sz; lo += sz+sz) {
            merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
        } //for
    } //for
} //sort
Bottom-Up Mergesort

• Rather than follow a tree-like structure, instead let’s merge all of the 1-length arrays into 2s in one pass, then all of the 2-length arrays into 4s, and repeat.
  – The second subarray may be smaller than the first in the last pass, but that’s no problem for `merge()` as we designed it.
**Bottom-Up Mergesort Visual**

<table>
<thead>
<tr>
<th>a[i]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>MERGESORT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXAMPLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- sz = 1
  - merge(a, 0, 0, 1)
  - merge(a, 2, 2, 3)
  - merge(a, 4, 4, 5)
  - merge(a, 6, 6, 7)
  - merge(a, 8, 8, 9)
  - merge(a, 10, 10, 11)
  - merge(a, 12, 12, 13)
  - merge(a, 14, 14, 15)

- sz = 2
  - merge(a, 0, 1, 3)
  - merge(a, 4, 5, 7)
  - merge(a, 8, 9, 11)
  - merge(a, 12, 13, 15)

- sz = 4
  - merge(a, 0, 3, 7)
  - merge(a, 8, 11, 15)

- sz = 8
  - merge(a, 0, 7, 15)
Bottom-Up Mergesort Evaluation

• Did our number of compares and array accesses change with this new implementation?
  – Well, we still have \( N \times \log(N) \) compares (\( \log(N) \) passes through as we increase \( sz \), and between \( N/2 \) and \( N \) items compared on each pass) and \( 6N \) array accesses per compare.
  – The exact number may have changed, but the order of growth has not.

• Bottom-up is better for linked lists. (why?)
Any Questions?