Sorting
Sorting

- A little old estimate said that more than half the time on many commercial computers was spent in sorting.
- Knuth’s book lists about 25 sorting methods and claims they are only fraction of the algorithms that have been devised so far.

- Types of sorting:
  - External vs. Internal

Insertion Sort

- Insertion in an Ordered List
Sorting by Insertion

- Maintain two lists, one sorted, another unsorted.
- Initially the sorted list has size zero, unsorted list has all the original keys.
- One by one insert the keys from unsorted list to the right position in the sorted list.

Select 6 Names and play contiguous and linked list versions! (Volunteer needed!)

Sorting by Insertion (Example)
Insertion Sort (contiguous list)

```c
void InsertionSort(List *list)
{
    Position fu;   /* first unsorted entry position */
    Position place; /* searches sorted part of list */
    ListEntry current; /* holds entry temporarily */

    for (fu = 1; fu < list->count; fu++)
        if (LT(list->entry[fu].key, list->entry[fu-1].key))
            current = list->entry[fu];

        for (place = fu - 1; place >= 0; place --) {
            list->entry[place+1]=list->entry[place];
            if (place==0 || LE(list->entry[place-1].key, current.key))
                break;
        }
        list->entry[place] = current;
}
```

Insertion Sort (linked list)

```c
void InsertionSort(List *list)
{
    ListNode *fu;   /* the first unsorted node to be inserted */
    ListNode *ls;   /* the last sorted node (tail of sorted sublist */
    ListNode *current, *trailing;
    if (list->head) {
        ls = list->head; /* An empty list is already sorted. */
        while (ls->next) {
            fu = ls->next; /* Remember first unsorted node. */
            if (LT(fu->entry.key, list->head->entry.key)) {
                ls->next = fu->next; fu->next = list->head; list->head = fu;
                /* Insert first unsorted at the head of sorted list. */
            } else {
                /* Search the sorted sublist. */
                trailing = list->head;
                for (current = trailing->next; GT(fu->entry.key, current->entry.key);
                     current = current->next)
                    trailing = current;
                /* First unsorted node now belongs between trailing and current. */
                if (fu == current)
                    ls = fu;
                else {
                    ls->next = fu->next; fu->next = current; trailing->next = fu;
                }
            }
        }
    }
}
```
Analysis

- The $i^{th}$ entry requires anywhere between 0 to $(i-1)$ iterations. On the average it requires
  \[ \frac{0+1+\ldots+(i-1)}{i-1} = \frac{i}{2} \text{ iterations} \]
- Each iteration has
  - 1 comparison and
  - 1 assignment
- Outside the loop there are
  - 1 comparison and
  - 2 assignments
  - cost is \( \text{Comp} = \frac{i}{2} + 1 \)
  \[ \text{Assignments} = \frac{i}{2} + 2 \]
- $i$ iterates from 2 to $n$.

### Insertion Sort

```c
void InsertionSort(List *list)
{
    Position fu;       /* first unsorted entry position*/
    Position place;    /* searches sorted part of list*/
    ListEntry current; /* holds entry temporarily*/

    for (fu = 1; fu < list->count; fu++)
    {
        if (LT(list->entry[fu].key, list->entry[fu-1].key))
        {
            current = list->entry[fu];
            place = fu - 1; place >= 0; place--;
            list->entry[place+1]=list->entry[place];
            if (place==0) /* Less than list->entry[place-1] */
                break;
            list->entry[place]=current;
        }
    }
}
```

- But before we proceed lets simplify using Big-O rules:
  \[
  \text{Comp} = \frac{i}{2} + O(1) \\
  \text{Assign} = \frac{i}{2} + O(1)
  \]

- Total Cost:
  \[
  = \sum_{n=2}^{n} \left( \frac{i}{2} + O(1) \right) = \frac{1}{2} \sum_{n=2}^{n} i + O(n) = \frac{1}{4} n^2 + O(n)
  \]
Quiz:
When the worst case performance occurs?
When the best case performance occurs?

Comments on Insertion Sort

- Insertion sort is an excellent method to check if a sorted list is still sorted.
- It is also good if a list is nearly in order.
- The main disadvantage of insertion sort is that there are too many moves, even on sorted keys, if just one key is out of place.
- A data which needs to travel at far away location needs to go through many steps.
- One data moves just one position in one iteration.
Selection Sort

- Selection sort one by one selects the max (or min) keys from the unsorted list and just appends them at the end of the sorted list.

- Consequently, there is no insertion cost.

![Diagram of Selection Sort]

Selection Sort (Contiguous list)

```c
void SelectionSort(List *list)
{
    Position current; /*position of place being correctly filled*/
    Position max; /*position of largest remaining key */
    for (current = list->count - 1; current > 0; current--)
    {
        max = MaxKey(0, current, list);
        Swap(max, current, list);
    }
}
```
Selection Sort (Contiguous list)

```c
Position MaxKey(Position low, Position high, List *list)
|
  Position largest; /* position of largest key so far */
  Position current; /* index for the contiguous list */
  largest = low;
  for (current = low + 1; current <= high; current++)
    if (LT(list->entry[largest].key, list->entry[current].key))
      largest = current;
  return largest;
|

void Swap(Position low, Position high, List *list)
|
  ListEntry temp = list->entry[low];
  list->entry[low] = list->entry[high];
  list->entry[high] = temp;
|
```

Analysis

- **Swap** is called \( n-1 \) times
  - each has 3 assignments
- **MaxKey** is called \( n-1 \) times. Length \( t \) of the sub list varies from \( n \) to 2.
  - Each requires \( t-1 \) comparisons.
  - Total 3(\( n-1 \)) assignments.
- **Thus there are:**
  - Thus \( (n-1)+(n-2)+\ldots+1 \)
  - \( = \frac{1}{2} n^2 + O(n) \)
Comparison of Selection and Insertion Sort

<table>
<thead>
<tr>
<th>Selection</th>
<th>Insertion (average)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignments of entries</td>
<td>$3.0n + O(1)$</td>
</tr>
<tr>
<td>Comparisons of keys</td>
<td>$0.5n^2 + O(n)$</td>
</tr>
<tr>
<td></td>
<td>$0.25n^2 + O(n)$</td>
</tr>
</tbody>
</table>

- Quiz:
- What is the best case for selection sort?
- What is the worst case for selection sort?
- Which method should we use
  - For large n?
  - If we know, the list is almost sorted?
  - Cost of assignment is large?

Shell Sort

- The problem with insertion sort is that, if a data needs to move much long distance it have to go through many iterations.
- Solution is Shell Sort!
- Invested by D.L. Shell in 1959.
Idea (Step-1)

Step-2
Shell Sort

- How to select the increments?
  - 5, 3, 1 worked. Many other choices will work also.
- However, no study so far could conclusively prove one choice is better than the other.
- Only requirement is that last round should be of increment 1 (that’s an insertion sort).
- Probably it isn’t a good idea to use increments in power’s of 2. Why?

- Analysis:
  - exceedingly difficult
  - for large $n$, it appears the number of moves is in $n^{1.25}$ to $1.6n^{1.25}$.

Lower Bounds of Sorting
Comparison Tree of Insertion Sort (a,b,c)

- The worst path is the worst case performance.
- The average path is the average performance.

Comparison Tree of Selection Sort (a,b,c)

- Selection sort tree is more bushy on the average.
Limits of Sorting Algorithms

- If there are $n$ numbers to sort how many possible outcomes?

**Theorem 7.2** Any algorithm that sorts a list of $n$ entries by use of key comparisons must, in its worst case, perform at least $\lceil \lg n! \rceil$ comparisons of keys, and, in the average case, it must perform at least $\lg n!$ comparisons of keys.

- Sterling’s approximation of $n!$:

  \[
  \log n! = \left( n + \frac{1}{2} \right) \log n - \left( \log e \right) n + \log \sqrt{2\pi} + \frac{\log e}{12n}
  \]

  \[
  \log n! = \left( n + \frac{1}{2} \right) \left( \log n - 1.5 \right) + 2
  \]

  \[
  = n \log n - 1.44n + O(\log n)
  \]

Next Class:
Quick & Merge Sort