Bucket Sort

Idea

- Counting sort assumes that the input consists of integers in a small range.
- Bucket sort assumes that the inputs are generated by a random process and elements are uniformly distributed over the interval [0,1].

- Algorithm:
  - Throws the numbers in their right buckets.
  - Sort each bucket with regular insertion sort.
  - Concatenate the buckets.
## Example

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.78</td>
</tr>
<tr>
<td>2</td>
<td>.17</td>
</tr>
<tr>
<td>3</td>
<td>.39</td>
</tr>
<tr>
<td>4</td>
<td>.26</td>
</tr>
<tr>
<td>5</td>
<td>.72</td>
</tr>
<tr>
<td>6</td>
<td>.94</td>
</tr>
<tr>
<td>7</td>
<td>.21</td>
</tr>
<tr>
<td>8</td>
<td>.12</td>
</tr>
<tr>
<td>9</td>
<td>.23</td>
</tr>
<tr>
<td>10</td>
<td>.68</td>
</tr>
</tbody>
</table>

12 → 17 → 23 → 26
68 → 72 → 78

## Code

```
BUCKET-SORT(A)
1. n ← length[A]
2. for i ← 1 to n
3. do insert A[i] into list B[[n/4][i]]
4. for i ← 0 to n - 1
5. do sort list B[i] with insertion sort
6. concatenate the lists B[0], B[1], ..., B[n - 1] together in order
```
Proof of Correctness

- If two items are in the same bucket then they are in proper relative order.
- If two items are in two different buckets, even then they are in right order.

Complexity

- Each small list is sorted with insertion sort:
  \[ \sum_{i=0}^{n-1} O(E[n_i^2]) = O \left( \sum_{i=0}^{n-1} E[n_i^2] \right) \]
- What is the expected length of the small lists?
  - Probability of a list to have length \(1\), \(2\), \(\ldots\), \(n\)?
  - Binomial distribution:
    \[ p = \frac{1}{n}, E[n_i] = np = 1, Var[n_i] = npq = 1 - \frac{1}{n} \]
    \[ E[n_i^2] = Var[n_i] + E^2[n_i], \]
    \[ = (1 - \frac{1}{n}) + 1^2 = 2 - \frac{1}{n} = \Theta(1) \]
    \[ \sum_{i=0}^{n-1} O(E[n_i^2]) = O \left( \sum_{i=0}^{n-1} E[n_i^2] \right) = O(n) \]
Class Mechanics

- Discussion about Final Project
- Feedback on Grade
- Quiz

Heap Sort
Motivation

- For Contiguous array, the best is QuickSort. Quicksort has no O(n log n) worst case bound. HeapSort has worst case bound O(n log n) and sorts in place.

Heap

**Definition** A heap is a list in which each entry contains a key, and, for all positions \( k \) in the list, the key at position \( k \) is at least as large as the keys in positions \( 2k \) and \( 2k + 1 \), provided these positions exist in the list.
Idea

- Build the heap,
  - by one by one inserting the keys.
- One by one take the roots out of the tree.
  - Insert it in the sorted list.
  - After each deletion rearrange the heap so that the largest again reaches at the top.

Example
Example (contd..)

Main & BuildHeap Code

```c
void HeapSort(List *list)
{
    Position lu;            /* Entries beyond lu have been
                            sorted. */
    ListEntry current;      /* holds entry temporarily
                            removed from list */
    BuildHeap(list);        /* First phase: turn list into
                            a heap. */
    for (lu = list->count - 1; lu >= 1; lu--) {
        current = list->entry[lu]; /* Extract last element
                                    from list */
        list->entry[lu] = list->entry[0]; /* Move top of
                        heap to end of list. */
        InsertHeap(current, 0, lu - 1, list);
    }
}

void BuildHeap(List *list)
{
    Position low;            /* Entries beyond low form a
                        heap. */
    for (low = list->count / 2 - 1; low >= 0; low--)
        InsertHeap(list->entry[low], low, list->count,
                   list);
}
```

Second half of the nodes already satisfies the heap condition.
InsertHeap

```c
void InsertHeap(ListEntry current,
                Position low, Position high, List *list)
{
    Position large;
    large = 2 * low + 1;
    while (large <= high) {
        if (large < high && LT(list->entry[large].key,
                              list->entry[large + 1].key))
            large++;
        if (GE(current.key, list->entry[large].key))
            break;
        else {
            list->entry[low] = list->entry[large];
            low = large;
            large = 2 * low + 1;
        }
    }
    list->entry[low] = current;
}
```

Analysis

- Each insertion may check log n heap nodes.
- Each check has two comparisons and one assignment.
- Let m=n/2, k varies from m-1 to 0. Cost of building is:
  \[ 2 \sum_{k=0}^{m-1} \log \frac{n}{k} = 2(m \log n - \log m!) = 5m = 2.5n \]
  \[ \log m = \log n - 1 \]
  \[ \log m! = m \log m - 1.5m = m \log m - 2.5m \]
- For sorting n elements we need to Insert (or restore the heap) n times. The cost is:
  \[ 2 \sum_{k=2}^{n} \log k \leq 2n \log n \]
- The worst case complexity is 2n log n comparisons
- and n log n assignments.
Heap Sort and Quick Sort

- Worst-case performance of Heap Sort (2nlogn) is poorer than the average-case performance of Quick Sort (1.39nlogn).
- However, the worst-case of Quick Sort is far worse than that of Heap Sort.
- The average-case analysis of Heap Sort is quite complex, however it shows it is almost same as its worst-case.
- On the average, therefore Quick Sort runs almost twice as fast as Heap Sort.