• Internal (external)  
  – All data are held in primary memory during the sorting process.

• Comparison Based  
  – Based on pairwise comparisons.  
  – Lower bound on running time is $\Omega(n\log n)$.

• In Place  
  – Requires very little, $O(\log n)$, additional space.

• Stable  
  – Preserves the relative ordering of items with equal values.
• $\Omega(n)$ to examine all the input.
• All sorts seen so far are $\Omega(n \lg n)$.
• We’ll show that $\Omega(n \lg n)$ is a lower bound for comparison sorts.
Lower Bounds for Comparison Sorts

• All possible flows of any comparison sort can be modeled by a decision tree.

• For example, the decision tree of insertion sort of 3 elements looks like:

• There are at least $n!$ leaves, because every permutation appears at least once.
Lower Bounds for Comparison Sorts

• Theorem:
  – Any decision tree that sorts $n$ elements has height $\Omega(n \lg n)$.

• Proof:
  – The number of leaves $l \geq n!$
  – Any binary tree of height $h$ has $\leq 2^h$ leaves.
  – Hence, $n! \leq l \leq 2^h$
  – Take logs: $h \geq \lg(n!)$
  – Use Stirling’s approximation: $n! > (n/e)^n$
  – We get:
    \[
    h \geq \lg(n/e)^n \\
    = n \lg(n/e) \\
    = n \lg n - n \lg e \\
    = \Omega(n \lg n).
    \]
Lower Bounds for Comparison Sorts

• Lemma:
  – Any binary tree of height \( h \) has \( \leq 2^h \) leaves.

• Proof:
  – By induction on \( h \).
    – Basis:
      • \( h = 0 \).
      • Tree is just one node, which is a leaf.
      • \( 2^h = 1 \).
    – Inductive step:
      • Assume true for height \( = h - 1 \).
      • Reduce all leaves and get a tree of height \( h - 1 \)
      • Make the tree full
      • Then, extend the tree of height \( h - 1 \) by adding to each leaf two son leaves.
      • You get a tree with at least many leaves as the original tree
      • Each leaf becomes parent to two new leaves.
      • \( \# \text{ of leaves for height } h = 2 \cdot (\# \text{ of leaves for height } h - 1) \)
        \( \leq 2 \cdot 2^{h-1} \) (ind. hypothesis)
        \( = 2^h \).
Sorting in Linear Time

- Non-comparison sorts.
- Counting Sort
- Radix Sort
- Bucket Sort
Counting Sort

• Input:
  – $A[1 \ldots n]$, where $A[j] \in \{0, \ldots, k\}$ for $j = 1, \ldots, n$.
  – Array $A$ and values $n$ and $k$ are given as parameters.

• Output:
  – $B[1 \ldots n]$, sorted.
  – $B$ is assumed to be already allocated and is given as a parameter.

• Auxiliary storage:
  – $C[0 \ldots k]$
Counting Sort

- **Complexity**: $\Theta(n + k)$, which is $\Theta(n)$ if $k = O(n)$.
Counting Sort

• How big a k is practical?
  – 32-bit values? No.
  – 16-bit? Probably not.
  – 8-bit? Maybe, depending on n.
  – 4-bit? Yes (unless n is really small).
Counting Sort

- Counting sort is **stable**
  - Keys with same value appear in same order in output as they did in input
  - Because of how the last loop works.
Radix Sort

radixSort \((A, d)\)

for \((i = 1 \text{ to } d)\) do

use a stable sort to sort array \(A\) on digit \(i\)
Radix Sort

• **Complexity:**

  – Assume that we use counting sort as the intermediate sort.
  – $\Theta(n + k)$ per pass (digits in range $0, \ldots , k$).
  – $d$ passes.
  – $\Theta(d(n + k))$ total.
  – If $k = O(n)$, time = $\Theta(dn)$. 
Radix Sort

• **Notations:**
  – n words.
  – b bits/word.
  – Break into r-bit digits.
    • Have \( d = \lceil b/r \rceil \).
  – Use counting sort, \( k = 2^r \).
  – Example:
    • 32-bit words, 8-bit digits.
    • \( b = 32, r = 8, d = 32/8 = 4, k = 2^8 = 256 \).

• **Complexity:**
  – \( \Theta((b/r) \cdot (n + 2^r)) \).
Radix Sort

• How to choose \( r \)?
  – Balance \( b/r \) and \( n + 2^r \).
  – Choosing \( r \approx \log n \) gives us \( \Theta(b/\log n \cdot (n + n)) = \Theta((b/\log n) \cdot n) \).
  – If we choose \( r < \log n \), then
    • \( b/r > b/ \log n \), and \( n + 2^r \) term doesn’t improve.
  – If we choose \( r > \log n \), then
    • \( n + 2^r \) term gets big.
    • For instance: \( r = 2 \log n \Rightarrow 2^r = 2^{2 \log n} = (2^{\log n})^2 = n^2 \).

• Example:
  – To sort \( 2^{16} \) 32-bit numbers,
  – Use \( r = \log 2^{16} = 16 \) bits.
  – Get \( b/r = 2 \) passes.
Radix Sort

• Compare radix sort to merge and quicksort:
  – 1 million \(2^{20}\) 32-bit integers.
  – Radix sort: \(32/20 = 2\) passes.
  – Merge sort/quicksort: \(\lg n = 20\) passes.

• Remember, though, that each radix sort “pass” is really 2 passes
  – one to take census, and one to move data.
Radix Sort

• Not a comparison sort:
  – We gain information about keys by means other than direct comparison of two keys.
  – Use keys as array indices.
Bucket Sort

• Assumes the input is
  – Generated by a random process that distributes elements uniformly over \([0, 1)\).

• Then
  – Divide \([0, 1)\) into \(n\) equal-sized buckets.
  – Distribute the \(n\) input values into the buckets.
  – Sort each bucket.
  – Go through buckets in order, listing elements in each one.
Bucket Sort

• Input:
  – \( A[1 \ldots n] \), where \( 0 \leq A[i] < 1 \) for all \( i \).

• Auxiliary array:
  – \( B[0 \ldots n – 1] \) of linked lists, each list initially empty.
Bucket Sort

```
BUCKET-SORT(A)
1   let B[0...n-1] be a new array
2   n = A.length
3   for i = 0 to n-1
4       make B[i] an empty list
5   for i = 1 to n
6       insert A[i] into list B[[n A[i]]]
7   for i = 0 to n-1
8       sort list B[i] with insertion sort
9   concatenate the lists B[0], B[1], ... , B[n-1] together in order
```
Bucket Sort

• Complexity:
  – All lines of algorithm except insertion sorting take \( \Theta(n) \) altogether.
  – If each bucket gets \( O(1) \) elements, the time to sort all buckets is \( O(n) \).
  – Hence, \( \Theta(n) \) altogether.

• Formally ...
Bucket Sort

- Define a random variable:
  - \( n_i \) = the number of elements placed in bucket \( B[i] \).
- Because insertion sort runs in quadratic time, bucket sort time is:

\[
T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)
\]

- Take expectations of both sides:

\[
E[T(n)] = E\left[ \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right]
\]

\[
= \Theta(n) + \sum_{i=0}^{n-1} E\left[ O(n_i^2) \right] \quad \text{(linearity of expectation)}
\]

\[
= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \quad \text{(E[\alpha X] = \alpha E[X])}
\]

\[
= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \quad \text{(E[n_i^2] = 2 - (1/n))}
\]

\[
= \Theta(n) + O(n)
\]

\[
= \Theta(n)
\]
Bucket Sort

• Claim:

\[ E[n_i^2] = 2 - \frac{1}{n} \] for \( i = 0, \ldots, n - 1 \).

• Proof:

– Define indicator random variables:

\[ X_{ij} = I\{ A[j] \text{ falls in bucket } i \} \]

\[ \Pr\{ A[j] \text{ falls in bucket } i \} = \frac{1}{n} \]

\[ n_i = \sum_{j=1}^{n} X_{ij} \]
Bucket Sort

• Then,

\[
E[n_i^2] = E \left[ \left( \sum_{j=1}^{n} X_{ij} \right)^2 \right] \\
= E \left[ \sum_{j=1}^{n} X_{ij}^2 + 2 \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} X_{ij} X_{ik} \right] \\
= \sum_{j=1}^{n} E[X_{ij}^2] + 2 \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} E[X_{ij} X_{ik}] \quad \text{(linearity of expectation)}
\]
Bucket Sort

• Now,

\[
E[X_{ij}^2] = 0^2 \cdot \Pr\{A[j] \text{ doesn’t fall in bucket } i\} + 1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\}
\]
\[
= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n}
\]
\[
= \frac{1}{n}
\]

• Since for \( j \neq k \), \( X_{ij} \) and \( X_{ik} \) are independent

\[
E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}]
\]
\[
= \frac{1}{n} \cdot \frac{1}{n}
\]
\[
= \frac{1}{n^2}
\]
Bucket Sort

• Therefore:

\[
E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + 2 \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \frac{1}{n^2}
\]

\[
= n \cdot \frac{1}{n} + 2 \left( \frac{n}{2} \right) \frac{1}{n^2}
\]

\[
= 1 + 2 \cdot \frac{n(n - 1)}{2} \cdot \frac{1}{n^2}
\]

\[
= 1 + \frac{n - 1}{n}
\]

\[
= 1 + 1 - \frac{1}{n}
\]

\[
= 2 - \frac{1}{n}
\]
Bucket Sort

• Again, **not a comparison sort**.
  – Used a function of key values to index into an array.

• **This is a probabilistic analysis.**
  – We used probability to analyze an algorithm whose running time depends on the distribution of inputs.

• Different from **a randomized algorithm**, where
  – We use randomization to impose a distribution.

• If the input isn’t drawn from a uniform distribution on \([0, 1)\),
  – All bets are off, but the algorithm is still correct.