Terminologies

- Comparison-based sorting
- In-place
- Stable sorting
- Internal and external sorting

Insertion Sort: complexity analysis

Worst-case

\[ T_{\text{worst}}(n) \leq \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \]

for loop to see whether E[i] needs to move

while loop to move E[i] at most i times to the front of the array

void quickSort(Element[] E, int first, int last)
if (first < last)
    Element pivot = E[first];
    Key pivot = pivot.key;
    int splitPoint = partition(E, pivot, first, last);
    quickSort(E, first, splitPoint - 1);
    quickSort(E, splitPoint + 1, last);
    return;

void insertionSort(Element[] E, int n)
for (int i = 1; i < n; i++)
    Element current = E[i];
    Key x = current.key;
    int slot = shiftVac(E, i, d);
    E[slot] = current;
    return;

int shiftVac(Element[] E, int index, Key x)
int vacant = index;
int vacLoc = 0; // Assume x is the smallest key, and will move to the leftmost position.
while (vacant > 0) { // still have elements to the left
    if (E[vacant - 1].key <= x) // if the element to the left has a smaller key
        vacant = vacant - 1; // here is the location to insert x.
        break;
    E[vacant] = E[vacant - 1]; // switch with the left neighbor that has a larger key
    vacant--; // and take your left neighbor's position.
} return vacLoc;

Insertion Sort
QuickSort: Partition

int partition(Element[] E, Key pivot, int first, int last)
1. low = first, high = last;
2. while (low < high)
3.   int higVac = extendLargeRegion(E, pivot, low, high);
4.   int lowVac = extendSmallRegion(E, pivot, low+1, highVac);
5.   low = lowVac; high = highVac - 1;
6. return low;

extendLargeRegion

int extendLargeRegion(Elemen[] E, key pivot, int low, int high)
1. higVac = low; // in case no key < pivot.
2. while (curr > low)
3.   if (E[curr].key < pivot) E[low] = E[curr];
4.   highVac = curr;
5.   break;
6. curr--; // keep looking.
7. return highVac;

extendSmallRegion

int extendSmallRegion(Elemen[] E, key pivot, int low, int high)
1. lowVac = high; // in case no key > pivot.
2. while (curr < high)
3.   if (E[curr].key >= pivot) E[high] = E[curr];
4.   lowVac = curr;
5.   break;
6. curr--; // keep looking.
7. return lowVac;

QuickSort: the partition subroutine

<table>
<thead>
<tr>
<th>Pivot</th>
<th>6</th>
<th>2</th>
<th>8</th>
<th>7</th>
<th>5</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

QuickSort: ideal-case

Each call to the partition subroutine will return a splitPoint which is right at the middle of the range, namely, divide the range into two equal subranges. In doing so, at most n comparisons are needed to ensure one subrange contains only keys that are smaller than the pivot, and the other subrange only keys larger than the pivot. Therefore,

\[ T(n) = 2T(n/2) + n \]

QuickSort: worst-case

Each call to the partition subroutine will return a splitPoint which is just the left boundary of the range, namely, no keys are smaller than the pivot. How many comparisons needed to know this? Still, all elements in the range need to compare with the pivot. When does this happen? Ironically when the array is a sorted one. In this case, divide-and-conquer decays into chip-and-conquer as the 2nd recursive call is quickSort(E, splitPoint + 1, last), chipping off one element from the range. Therefore,

\[ T(n) = T(n-1) + n \]
Assumption: \[ P(\text{splitPoint} = i | \text{splitPoint} = \text{partition}[1 \to n]) = \frac{1}{n} \]

Then:
\[ T(n) = n + \sum_{i=1}^{n-1} (\frac{1}{n} T(i) + \frac{1}{n} T(n-1-i)) \]
\[ = n + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \]

**Theorem 4.2.** \[ T(n) \leq c n \ln(n) \] holds for any \( n \), where \( c \) is a constant.

**Proof:** induction on \( n \).

**Base case** \( n = 1 \). One single element array is sorted, i.e., \( T(1) = 0 \). As \( c \cdot 1 \cdot \ln(1) = 0 \), the theorem holds for the base case.

Assume for any \( n \), \( T(n) \leq c n \ln(n) \), then:
\[ T(n+1) = (n+1) - 1 + \frac{2}{n+1} \sum_{i=1}^{n} (T(i) + T(n+1-i)) \]
\[ = n + \frac{2}{n+1} \sum_{i=1}^{n} T(i) + \frac{2}{n+1} \sum_{i=1}^{n} T(n+1-i) \]
\[ = n + \frac{2}{n+1} \sum_{i=1}^{n} T(i) + \frac{2}{n+1} \sum_{i=1}^{n} T(n+1-i) \]
\[ = n + \frac{2}{n+1} \sum_{i=1}^{n} T(i) + \frac{2}{n+1} \sum_{i=1}^{n} T(n+1-i) \]
\[ = n + \frac{2}{n+1} \sum_{i=1}^{n} T(i) + \frac{2}{n+1} \sum_{i=1}^{n} T(n+1-i) \]
\[ \leq 2(n+1) \ln(n+1) \]

QED

**Randomized quicksort**

Using a randomly chosen element as pivot will enforce the equal distribution assumption made at the average-case analysis.

**Mergesort**

void mergesort(Element[] E, int first, int last)
\( d = \text{(first + last) / 2; } \)
mergeSort(E, first, d); mergeSort(E, d+1, last);
return;

- Unlike quickSort, mergeSort guarantees equal division each time.
- Array with a single element is sorted.
- Work is done at the combining steps.

**Mergesort: Example**

![Mergesort Example](image)

**Mergesort: the merge subroutine**

merge(E, first, mid, last)
\( M[0 \to \text{last}] = \text{temp array; } \)
for \( k = \text{mid} + 1 \) to \( \text{last}; i++ \)
\( M[i-1] = E[i-1]; \)
for \( i = \text{mid}; j = \text{last}; j--; \)
\( E[k] = (M[i] < M[j]) ? M[i++] : M[j--]; \)
for \( k = \text{last}; k > \text{first}; k--; \)
\( E[k] = (M[i] < M[j]) ? M[i++] : M[j--]; \)

Note: extra space needed \( \Rightarrow \) not in-place sorting

**QuickSort: space complexity**

If \( k \) is “typical” sort, \( \text{not in-place sorting is needed.} \)
Yet there are hidden space usages, checks for recursive calls \( \Rightarrow O(1) \) for worst-case, and \( \ln(n) \) on average. The following modified algorithm guarantees this \( \Omega(n) \) usage of space.

```java
public void quickSort(int[] E, int first, int last, int k)
{ int mid = (first + last) / 2;
  quickSort(E, first, mid - 1, k);
  quickSort(E, mid + 1, last, \( k \));
}
```

```java
public void mergesort(Element[] E, int first, int last)
{ if (first < last)
  int mid = (first + last) / 2;
  mergesort(E, first, mid);
  mergesort(E, mid + 1, last);
  merge(E, first, mid, last);
}
```
Mergesort: analysis

- **Worst-case (Q: when?)**
  \[ T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n - 1 \]
  \[ T(1) = 0 \]
  Master Theorem \( \Rightarrow T(n) \in \Theta(n \log n) \)
  
  Note: 1\textsuperscript{st} algorithm by far does \( \log(n) \) for worst-case.
  Can we do better? Or will be better off on average?

---

Lower bounds: worst-case

- Decision tree approach
  - nodes \( \leftrightarrow \) comparisons of keys
  - leaves \( \leftrightarrow \) possible permutation of \( n \) keys (\( = n! \))

  ![Decision Tree Diagram]

  Height \( \leftrightarrow \) max \# of comparisons
  \( \geq \lg (\# \text{ of leaves}) = \lg(n!) \geq \frac{n}{2}\lg(n) \)
  \( \in \Theta(n \log n) \)

---

Lower bounds

- **Theorem 4.10** Any algorithms to sort \( n \) items by comparisons of keys must do at least \( \lg(n)! \), or approximately \( n \lg n - 1.443n \), key comparisons in the worst case.
  - Mergesort is optimal for the worst case.

- **Average-Case**
  - \( T_{av}(n) = \text{(sum of lengths of all paths from the root to a leaf) / (L, \# of leaves)} \)
  - Balanced decision tree \( \Rightarrow \) lower \( T_{av}(n) \)
  - A complete tree is most balanced: \( [L \lg(L)] / L \)
  - Therefore,
  \[ T_{av}(n) \geq \lg(L) = \lg(n!) \in \Theta(n \log n) \]