CISC 320 Introduction to Algorithms
Fall 2003

Lecture 3
Sorting

- Terminologies
  - Comparison-based sorting
  - In-place
  - Stable sorting
  - Internal and external sorting
void insertionSort (Element[] E, int n)
    for (int i = 1; i < n; i++)
        Element current = E[i];
        Key x = current.key;
        int xLoc = shiftVac(E, i, x);
        E[xLoc] = current;
    return ;

int shiftVac (Element[] E, int xindex, Key x)
    int vacant, xLoc;
    vacant = xindex;
    xLoc = 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) // the element to the left has smaller key
            xLoc = vacant; // 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 xLoc;
Insertion Sort: complexity analysis

- Worst-case

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

- 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

Quicksort

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

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

---

```c
int extendLargeRegion(Element[] E, key pivot, int low, int high)
    int highVac, curr;
    highVac = low; // in case no key < pivot.
    curr = high;
    while (curr > low)
        if (E[curr].key < pivot)
            E[low] = E[curr];
            highVac = curr;
            break;
        curr --; // keep looking.
    return highVac;
```

int extendSmallRegion(Element[] E, key pivot, int low, int high)
    int lowVac, curr;
    highVac = high; // in case no key < pivot.
    curr = low;
    while (curr < high)
        if (E[curr].key >= pivot)
            E[high] = E[curr];
            lowVac = curr;
            break;
        curr --; // keep looking.
    return lowVac;

Quicksort: the partition subroutine

Pivot       low    6 2 8 7 5 3 1
            1 6 2 8 7 5 3
            1 2 8 7 5 3 6
            1 3 2 8 7 5 6
            1 3 2 7 5 8 6
splitPoint

≤ Pivot   3 2 , 4 7 5 8 6 ≥ Pivot

CISC320, Fall, Liao, Liao
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,

$$T(n) = T(n-1) + n$$
QuickSort: Average-case

Assumption: \[ \text{Prob}(\text{splitPoint} = i | \text{splitPoint} = \text{partition}[1 \text{ to } n]) = \frac{1}{n} \]

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

**Theorem 4.2** \(T(n) \leq cn \ln(n)\) holds for any \(n \geq 1\), 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 \(c \ln(1) = 0\), the theorem holds for the base case.

Assume for any \(n\), \(T(n) \leq cn \ln(n)\), then
\[T(n+1) = (n+1)-1 + \frac{2}{(n+1)} \sum_{i=1}^{n} T(i)
\]
\[\leq n + \frac{2c}{(n+1)} \int_1^{n} x \ln(x) \, dx
\]
\[= n + 2c \left[ \frac{1}{2} (n+1)^2 \ln(n+1) - \frac{1}{4} (n+1)^2 + \frac{1}{4} \right] / (n+1)
\]
\[= C(n+1) \ln(n+1) + (n+1) \left[ 1 - \frac{c}{2} \right] + \left[ \frac{c}{2} (n+1)^2 \right] - 1
\]
\[\leq 2(n+1) \ln(n+1) \quad \text{if } c = 2
\]

QED

Randomized quicksort

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

It is “in-place” sort, as no extra array is needed. Yet there are hidden space usage: stacks for recursive calls -> $O(n)$ for worst case, and $\ln(n)$ on average. The following modified algorithm guarantees this $\lg(n)$ usage of space.

```c
void quickSortTRO(Element[] E, int first, int last)
    int first1, last1, first2, last2;
    first2 = first; last2 = last;
    while (last2 - first2 >= 1)
        pivotElement = E[first2];
        pivot = pivotElement.key;
        int splitPoint = partition(E, pivot, first2, last2);
        E[splitPoint] = pivotElement;
        if (splitPoint <= (first2 + last2) / 2)
            first1 = first2; last1 = splitPoint - 1;
            first2 = splitPoint + 1; last2 = last2;
        else
            first1 = splitPoint + 1; last1 = last2;
            first2 = first2; last2 = splitPoint - 1;
        quickSortTRO(E, first1, last1);
    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: the merge subroutine

merge(E, first, mid, last)
  int i, j;
  Element[] M; // temp array.
  for (i=mid + 1; i>first; i--)
    M[i-1] = E[i-1];
  for (j = mid; j<last; j++)
    M[last + mid -j] = E[j+1];
  for (k=first; k <= last; k++)
    E[k] = (M[i] < M[j]) ? M[i++] : M[j--];

Note: extra space needed => not in-place sorting
Mergesort: analysis

- Worst-case (Q: when?)
  \[ T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1 \]
  \[ T(1) = 0 \]
  Master Theorem \(\Rightarrow\) \(T(n) \in \Theta(n \log n)\)
  Note: 1st algorithm by far does \(n \log(n)\) 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](image)

  Height \(\leftrightarrow\) max # of comparisons
  \[ \geq \lg(\text{# of leaves}) = \lg(n!) \geq (n/2) \lg(n/2) \]
  \(\in \Theta(n \log n)\)
Theorem 4.10  Any algorithms to sort \( n \) items by comparisons of keys must do at least \( \Omega(\lg(n!)) \), or approximately \( \Gamma n \lg n - 1.443n \), key comparisons in the worst case.

- Mergesort is optimal for the worst case.

### Average-Case

- \( T_{av}(n) = \frac{\text{(sum of lengths of all paths from the root to a leaf)}}{\text{(L, # of leaves)}} \)
- Balanced decision tree => lower \( T_{av}(n) \)
- A complete tree is most balanced: \([L \lg(L)] / L\)
- Therefore,

\[
T_{av}(n) \geq \lg(L) = \lg(n!) \in \Theta(n \log n)
\]