SORTING

QuickSort

- A Divide-and-Conquer Algorithm
- Best and Worst cases
  - Analysis

Sorting – Divide and Conquer

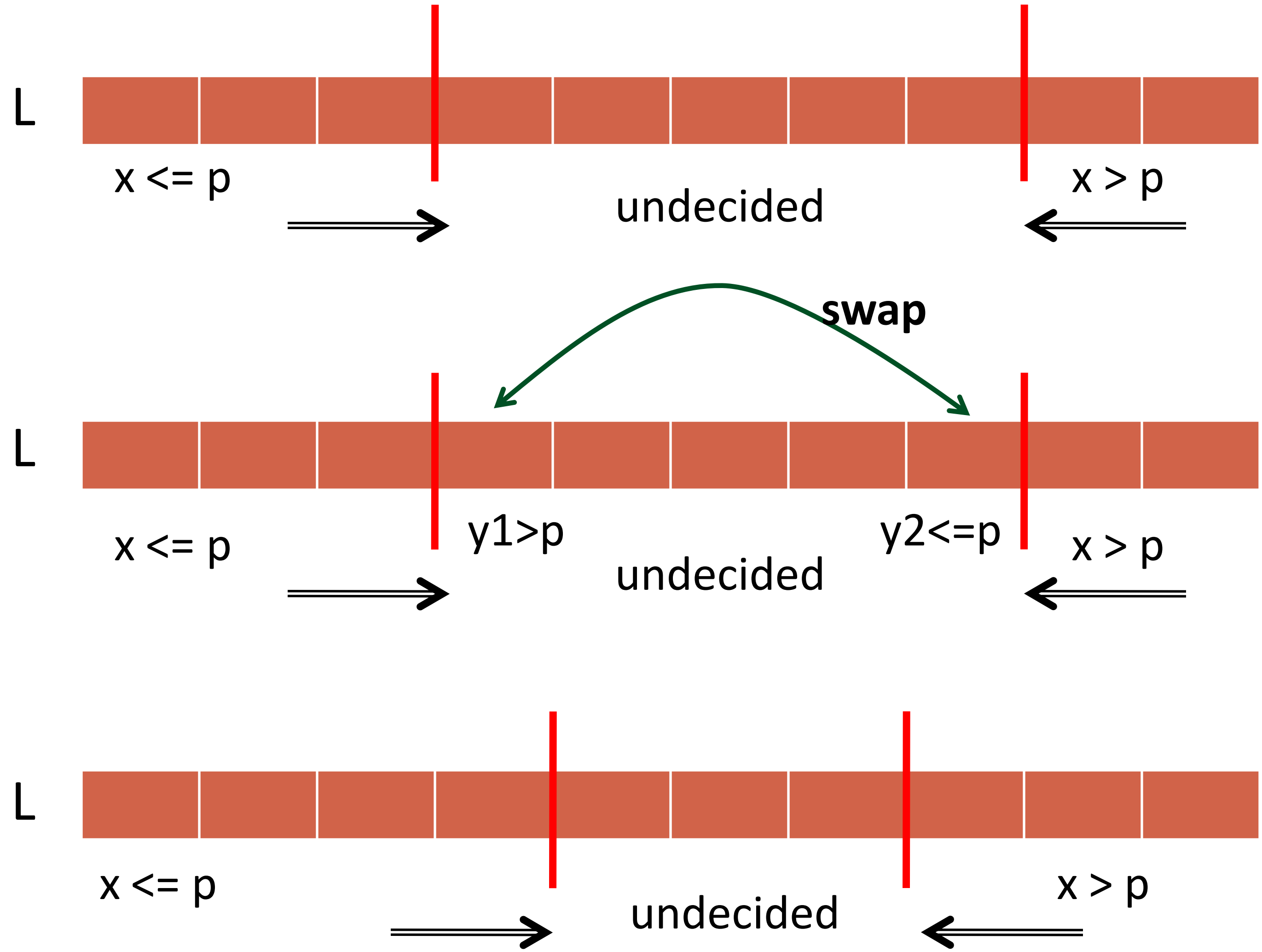
- The two sorting algorithms seen so far:
  - Insertion Sort and Merge Sort
- Both were designed by using:
  - divide-and-conquer as the design technique
  - an *order-preserving* combination operation (***insert*** or ***merge***)
- Question:
  - Are there other ways to divide-and-conquer?
    - Are there other ways of dividing problems / combining solutions ?

QuickSort

- Input: List of **N** elements **L[0], L[1], … L[N-1]**
- Divide and Conquer:
  - Partition **L** into **LL** and **LG** based on a pivot **p** in **L** s.t. :
    - **LL = { x in L | x <= p }**
    - **LG = { x in L | x > p }**
  - Combine sorted versions of **LL** and **LG** and **{p}**:
    - Append: **LL, { p }, LG**
      - Append is trivial if **LL** & **LG** were sorted in place and p is in position
      - Sorting in place requires partitioning in place:
        - Question: How do you keep **p** in between?

QUICKSORT

- (Hoare's) Partitioning : (pivot p in L)



QuickSort

Input: List of elements Ls[lo], … L[hi]

```
qs(Ls, lo, hi)
{
    if (lo<hi) {
        p=pivot(Ls,lo,hi);    // Ls[p] is the pivot
        p=part(Ls,lo,hi,p);   // Ls[p] is the pivot
        /* (Ls[j]<=Ls[p] for j in lo..pPos-1) and
            (Ls[j]>Ls[p] for j in pPos+1..hi)
        */
        qs(Ls,lo,p-1);
        qs(Ls,p+1,hi);
    }
}
```

QuickSort – Time Complexity

- Time Complexity of Partition:
  - $\Theta(N)$ - /* # comparisons? # swaps? */
- Quick Sort – Time - Worst case (recurrence relation):
  - $T(N) = T(N-1) + \Theta(N)$ for $N>1$
  - $= \Theta(1)$ for $N=1$
    - $T(N) = ?$
    - When does the worst case occur?
- Quick Sort – Time - Best case (recurrence relation):
  - $T(N) = 2*T(N/2) + \Theta(N)$ for $N>1$
  - $= \Theta(1)$ for $N=1$
    - $T(N) = ?$
    - When does the best case occur?

QUICKSORT – TIME COMPLEXITY

- Average Case:
  - By <u>assuming input distributions to be random</u> *one can compute the average case complexity.*
  - Verify that O(N*logN) is a solution to recurrence relation:
    - $T(N) = \Theta(N) + (1/N) * (\sum_{k=1 to N} (T(k-1) + T(N-k)))$
      for N>1
    - $T(N) = 1$ for $N <= 1$

- But the <u>time taken for a specific input</u> depends heavily <u>*on the pivot(s) chosen*</u> for partitioning.

QuickSort : Space Complexity

- What is the depth of recursion?
  - Worst Case: N-1
    - Why?
  - Best Case: $\log_2 N$
    - Why?

QUICKSORT - CALL STACK OVERHEAD

Recursive version

```
void  qs(Element ls[],
                 int lo, int hi)
{
  if (lo<hi) {
    p = pivot(ls,lo,hi);
    p=part(ls,lo,hi,p);
    qs(ls, lo, p-1);
    qSort(ls, p+1, hi);
   }
}
```

Tail call elimination

```
void qs(Element ls[],
              int  lo, int hi)
{
    while (lo<hi) {
      p=pivot(ls, lo, hi);
      p=part(ls, lo, hi, p);
      qs(ls, lo, p-1);
      lo = p+1;
     }
   }
```

QuickSort - Call Stack Overhead

Recursive version
(w/o tail calls)

```
void qs(Element ls[],
             int  lo, int hi)
{
    while (lo<hi) {
      p=pivot(ls, lo, hi);
      p=part(ls, lo, hi, p);
      qs(ls, lo, p-1);
      lo = p+1;
      }
    }
```

Iterative version (. explicit stack)

```
void qs(Element ls[], int lo, int hi)
  {
   s = push(newStack(), (lo,hi));
   while (!isEmpty(s)) {
    (lo,hi)=top(s);   s=pop(s);
     while (lo<hi) {
         p = pivot(ls, lo, hi, p);
         p = part(ls, lo, hi, p);
         s = push(s, (lo,p-1));
         lo = p+1;
       }
     }
  }
```

QUICKSORT – SPACE COMPLEXITY

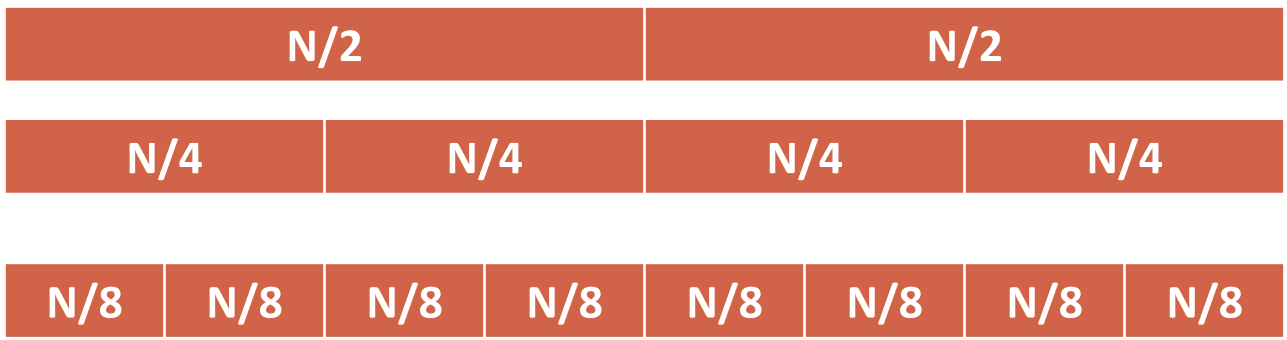
- 1. Avoid putting trivial lists on stack:
  - i.e. push only if *start* < *end*
- 2. Put the smaller of the two sub-lists on stack after each partitioning:
  - Every list is above a list that is (at least) twice as large
    - i.e. at most $\log_2 N$ items on stack at any point in time
      - where N is initial size.
- Exercise: Implement this version!

PARTITIONING

```
// Ls[lo..hi] is the input array; Ls[pInd] is the pivot
int partition(Ls, lo, hi,pInd)
{
    swap(Ls,pInd,lo);
    lt=lo+1;      rt=hi;        pv=Ls[lo];
    while (lt<rt) {
        for( ; lt<=hi && Ls[lt]<=pv; lt++) ;        // Ls[j]<=pv  for j in lo..lt-1
        for( ; Ls[rt]>pv; rt--) ;                   // Ls[j]>pv  for j in rt+1..hi
        if (lt<rt) { swap(Ls,lt,rt); lt++; rt--; }
    }
    if (lt==rt) pPos=lt; else pPos=lt-1;     // Do we need this? Is lt==rt possible?
    swap(Ls,lo,pPos);
    //Postcond.: (Ls[j]<=pv for j in lo..pPos-1) and (Ls[j]>pv for j in pPos+1..hi)
    return pPos;
}
```

CASE ANALYSIS

- Best case Partitioning:



…

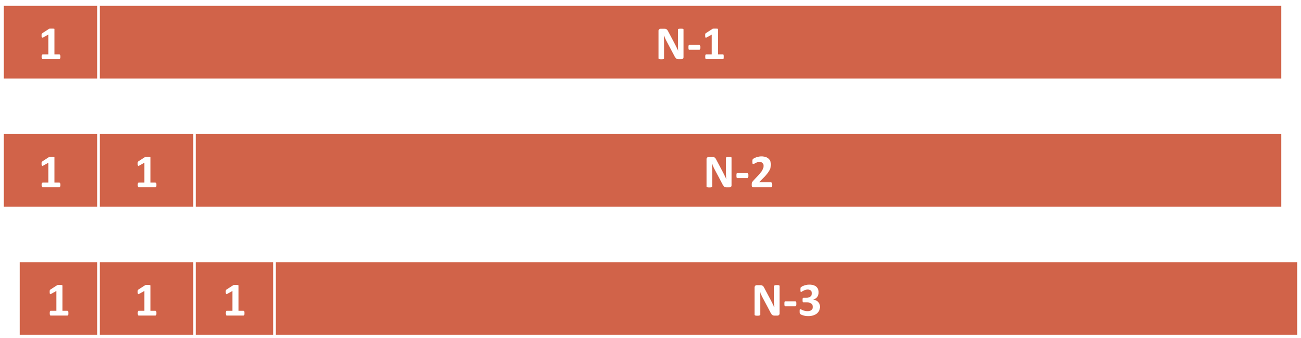
logN steps

Partitioning work in each step is O(N)

Time Complexity: O(NlogN)

CASE ANALYSIS

- Worst case Partitioning:



- …

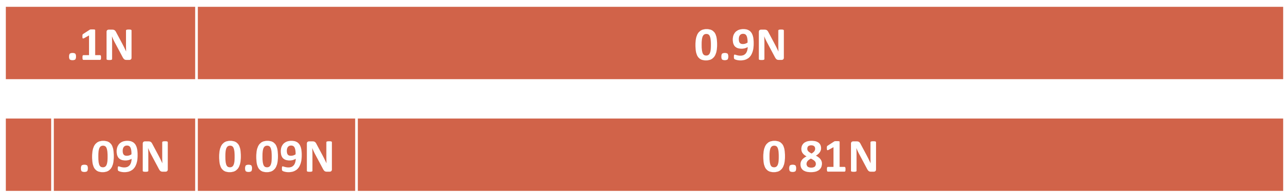
N-1 steps

Partitioning work in each step is O(N)

Time Comlexity: O(N*N)

Balanced Partitioning

- Better time complexity with balanced partitioning
- Consider the following example scenario:



...

- Time Complexity:

$T(N) = T(N/10) + T(9N/10) + \Theta(N)$

$<= 2*T(9N/10) + \Theta(N)$

$= O(N*\log_{10/9} (N))$

PIVOT SELECTION

- Biased pivots result in unbalanced partition
  - Pivot p such that p < x for most x in L
    - Symmetrically, p > x for most x in L
- Static/Fixed techniques for pivot selection repeat the bias at every level
  - E.g. First element of the list as a pivot in a (mostly) sorted list
- De-biasing Solution:
  - Adaptive selection of pivots
    - Typically done by sampling the input

Pivot Selection Techniques

- Several Options
  - Median of 3
    - This is still a fixed partitioning technique but it is a better sample than one location!
  - Median of Medians
  - QuickSelect (selecting Kth smallest element)
  - Random

- Cost of *partition* and cost of *pivot selection*
  - Pivot selection should not take more time than partition
    - i.e. pivot selection should be done in $\theta(N)$

Pivot Selection Techniques

- Median of 3:
  - Median of first, middle, and last element in the (sub)list
  - Exclude these elements from partitioning process

- Questions:
  - Will this improve the time complexity of QuickSort?
  - What is the minimum number of comparisons required for finding median of 3 values?

PIVOT SELECTION TECHNIQUES

- Median of Medians:
  - For every 5 contiguous elements find the median by direct comparison ==> N/5 medians
  - Obtain the median of these N/5 medians
    - How?
      - Sort?   Why is this a bad idea?
      - QuickSelect(Ls,N,N/2)?
- Assume
  - QuickSelect (Ls, N, K)
    - selects (and returns) the Kth smallest of N elements in Ls
  - *We will revisit this*.

- Impact of MoM:
  - Will this reduce the complexity of QuickSort?

PIVOT SELECTION - QUICKSELECT

- QuickSelect(Ls,lo,hi,K) // selecting Kth smallest element
  1. p = pivot(Ls,lo,hi); // Ls[p] is the pivot
  2. p = partition(Ls,lo,hi,p) // pivot is in its correct position p
  3. if K == p done,

     if K<p QuickSelect(Ls,lo,p-1,K);

     else QuickSelect(Ls,p+1,hi,K-p);

Q: What is the time complexity of QuickSelect?

Q: How do you ensure QuickSelect is done in optimal time?
i.e. How do you ensure the partition called in QuickSelect is balanced?

Pivot Selection

- Randomized QuickSort
  - Select pivot index uniformly randomly between first and last (indices).
  - Need a (good) random number generator
- Is there a random number?
  - Random sources
  - Bit Selection
    - Repeated coin toss
  - Cost of random number generation
  - Pseudo-random number generators
- Exercise:
  - Assuming random pivot selection (in each call to partition):
    - Estimate the probability of the worst case behavior of QuickSort.
    - Estimate the probability of the best case behavior of QuickSort.

Partitioning

```
// Ls[lo..hi] is the input array; Ls[pInd] is the pivot
int partition(Ls, lo, hi,pInd)
{
    swap(Ls,pInd,lo);
    lt=lo+1;     rt=hi;      pv=Ls[lo];
    while (lt<rt) {
        for( ; lt<=hi && Ls[lt]<=pv; lt++) ;        // Ls[j]<=pv  for j in lo..lt-1
        for( ; Ls[rt]>pv; rt--) ;                   // Ls[j]>pv  for j in rt+1..hi
        if (lt<rt) { swap(Ls,lt,rt); lt++; rt--; }
    }
    if (lt==rt) pPos=lt; else pPos=lt-1;
    swap(Ls,lo,pPos);
    //Postcond.: (Ls[j]<=pv for j in lo..pPos-1) and (Ls[j]>pv for j in pPos+1..hi)
    return pPos;
}
```

SMALL LISTS

- Insertion Sort performs better than QuickSort on small lists.
  - Why?
  - So, what?
- Combine the two!
  - Invoke Insertion Sort inside QuickSort when size of the list is small.
  - Time Complexity (expected):
    - O(k*k*(N/k) + N*log(N))
      - where k is the threshold (below which InsertionSort performs better than QuickSort)
- Alternatively, ignore, small sized lists inside QuickSort, and do an insertion Sort (on the full list)
    - Question: Why does this work (efficiently)?
    - Question: What is the time complexity?

EQUAL VALUES

- QuickSort performs badly when the same key occurs multiple times
- Solution: 3-way partition
  - Maintain an additional partition for elements equal to the pivot
    - Exercise: Implement this!