COMP9024: Data Structures and Algorithms

Week Eight: Sortings and Sets

Hui Wu

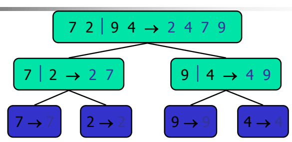
Session 1, 2015

http://www.cse.unsw.edu.au/~cs9024

Outline

- Merge Sort
- Quick Sort
- Bucket-Sort
- Radix Sort
- Sorting Lower Bound
- Union-Find Partition Structures

Merge Sort



Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1
- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It uses a comparator
  - It has $O(n \log n)$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm ***mergeSort(S, C)***
  Input sequence S with n elements, comparator C
  Output sequence S sort according to C

```
{
  if ( S.size() > 1 )
  {
    (S1, S2) = partition(S, n/2);
    mergeSort(S1, C);
    mergeSort(S2, C);
    S = merge(S1, S2);
  }
}
```


Merging Two Sorted Sequences

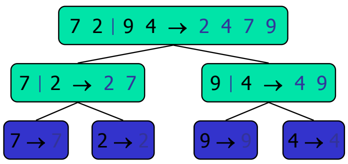
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

Algorithm ***merge(A, B)***
  Input sequences A and B with $n/2$ elements each
  Output sorted sequence of $A \cup B$

```
{
  S = empty sequence;
  while ( ¬A.isEmpty() ∧ ¬B.isEmpty() )
    if ( A.first().element() < B.first().element() )
      S.insertLast(A.remove(A.first()));
    else
      S.insertLast(B.remove(B.first()));
  while ( ¬A.isEmpty()
    S.insertLast(A.remove(A.first()));
  while ( ¬B.isEmpty() )
    S.insertLast(B.remove(B.first()));
  return S;
}
```

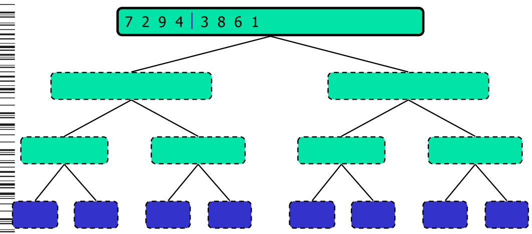
Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



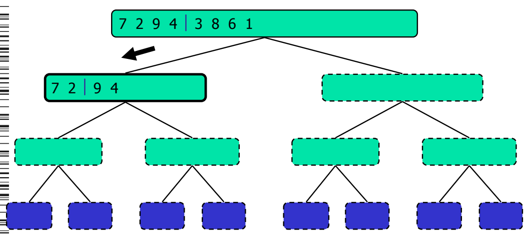
Execution Example (1/10)

- Partition



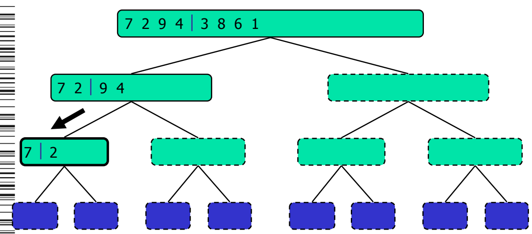
Execution Example (2/10)

- Recursive call, partition



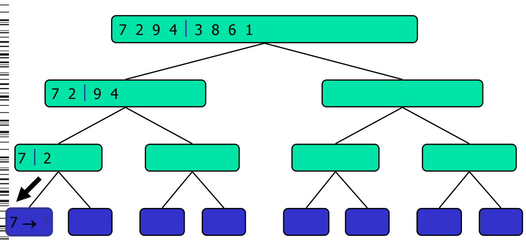
Execution Example (3/10)

- Recursive call, partition



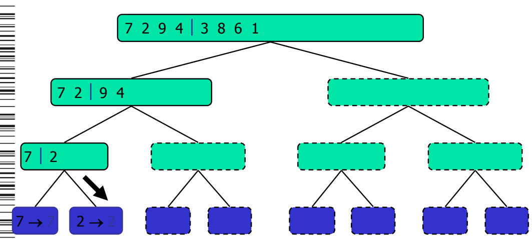
Execution Example (4/10)

- Recursive call, base case



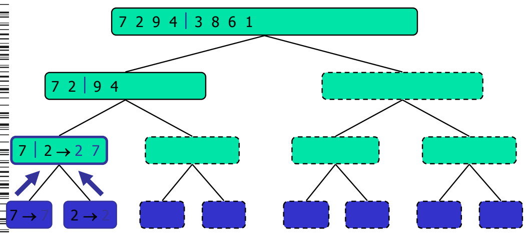
Execution Example (5/10)

- Recursive call, base case



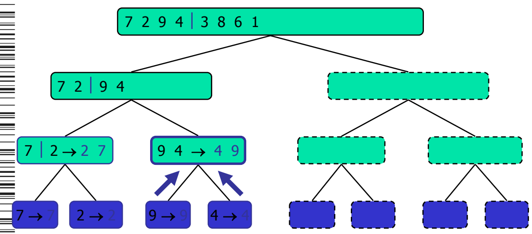
Execution Example (6/10)

- Merge



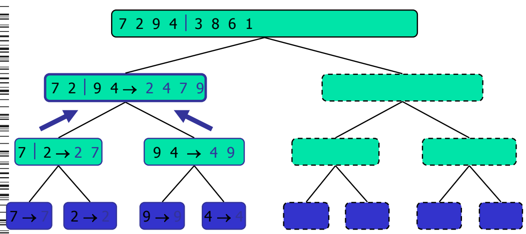
Execution Example (7/10)

- Recursive call, …, base case, merge



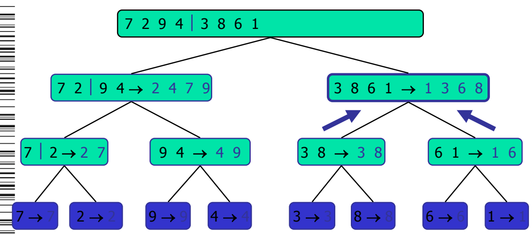
Execution Example (8/10)

- Merge



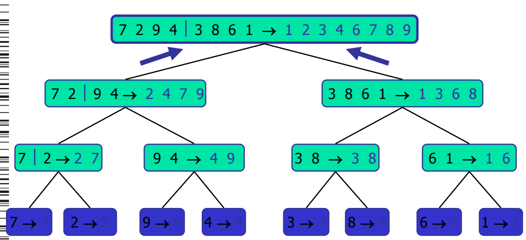
Execution Example (9/10)

- Recursive call, …, merge, merge



Execution Example (10/10)

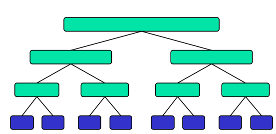
- Merge



Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount of work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	fast in-place for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	fast sequential data access for huge data sets (> 1M)

Nonrecursive Merge-Sort

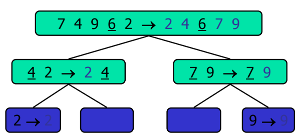
merge runs of length 2, then 4, then 8, and so on

```
public static void mergeSort(Object[] orig, Comparator c) { // nonrecursive
  Object[] in = new Object[orig.length]; // make a new temporary array
  System.arraycopy(orig,0,in,0,in.length); // copy the input
  Object[] out = new Object[in.length]; // output array
  Object[] temp; // temp array reference used for swapping
  int n = in.length;
  for (int i=1; i < n; i*=2) { // each iteration sorts all length-2*i runs
    for (int j=0; j < n; j+=2*i)  // each iteration merges two length-i pairs
      merge(in,out,c,j,i); // merge from in to out two length-i runs at j
    temp = in; in = out; out = temp; // swap arrays for next iteration
  }
  // the "in" array contains the sorted array, so re-copy it
  System.arraycopy(in,0,orig,0,in.length);
}
```

merge two runs in the in array to the out array

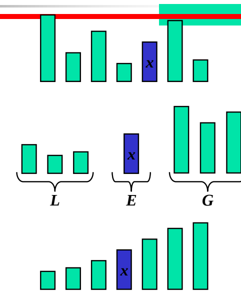
```
protected static void merge(Object[] in, Object[] out, Comparator c, int start,
    int inc) { // merge in[start..start+inc-1] and in[start+inc..start+2*inc-1]
  int x = start; // index into run #1
  int end1 = Math.min(start+inc, in.length); // boundary for run #1
  int end2 = Math.min(start+2*inc, in.length); // boundary for run #2
  int y = start+inc; // index into run #2 (could be beyond array boundary)
  int z = start; // index into the out array
  while ((x < end1) && (y < end2))
    if (c.compare(in[x],in[y]) <= 0) out[z++] = in[x++];
    else out[z++] = in[y++];
  if (x < end1) // first run didn't finish
    System.arraycopy(in, x, out, z, end1 - x);
  else if (y < end2) // second run didn't finish
    System.arraycopy(in, y, out, z, end2 - y);
}
```


Quick-Sort



Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



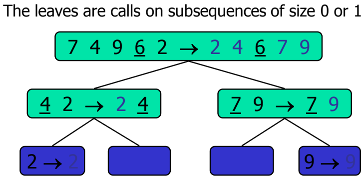
Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

Algorithm *partition*(S, p)
 Input sequence S, position p of pivot
 Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
 { L, E, G = empty sequences;
  $x = S.remove(p)$;
  while ($\neg S.isEmpty()$)
   { $y = S.remove(S.first())$;
    if ($y < x$)
     $L.insertLast(y)$;
    else if ($y = x$)
     $E.insertLast(y)$;
    else // $y > x$
     $G.insertLast(y)$;}
  return L, E, G;
 }

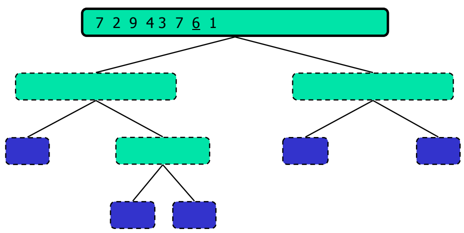
Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1



Execution Example (1/7)

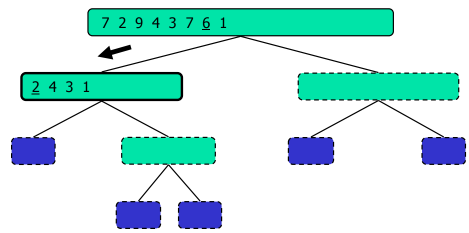
- Pivot selection



25

Execution Example (2/7)

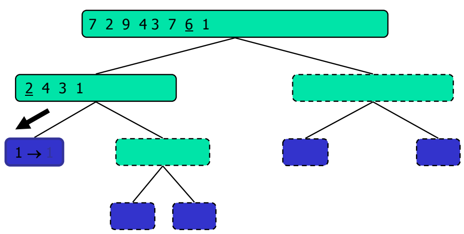
- Partition, recursive call, pivot selection



26

Execution Example (3/7)

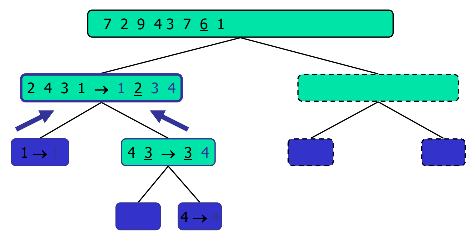
- Partition, recursive call, base case



27

Execution Example (4/7)

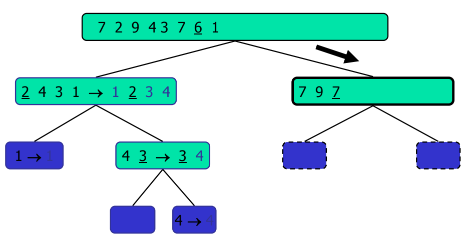
- Recursive call, …, base case, join



28

Execution Example (5/7)

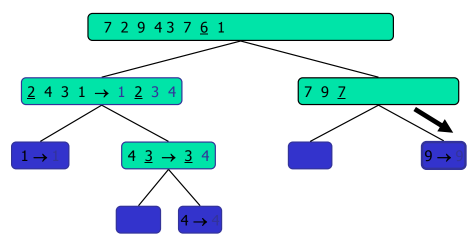
- Recursive call, pivot selection



29

Execution Example (6/7)

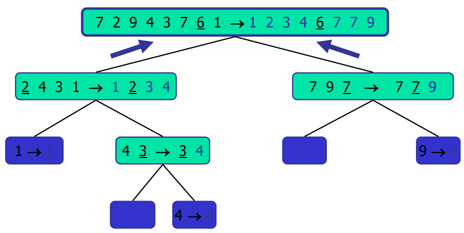
- Partition, …, recursive call, base case



30

Execution Example (7/7)

- Join, join

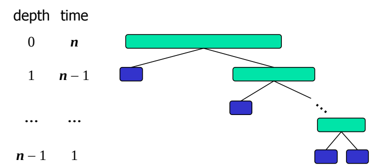


Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n-1$ and the other has size 0
- The running time is proportional to the sum

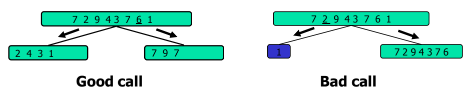
$$n + (n-1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $O(n^2)$

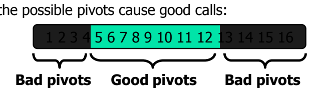


Expected Running Time (1/2)

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$

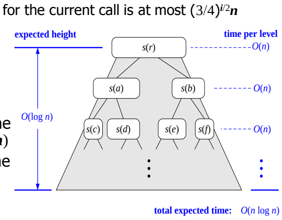


- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



Expected Running Time (2/2)

- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$
- For a node of depth i, we expect
  - $i/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$
- Therefore, we have
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



In-Place Quick-Sort

- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than h
  - the elements equal to the pivot have rank between h and k
  - the elements greater than the pivot have rank greater than k
- The recursive calls consider
  - elements with rank less than h
  - elements with rank greater than k

Algorithm *inPlaceQuickSort*(S, l, r)
  Input sequence S, ranks l and r
  Output sequence S with the elements of rank between l and r rearranged in increasing order
{ **if** ($l \geq r$)
    return;
  i = a random integer between l and r;
  $x = S.elemAtRank(i)$;
  $(h, k) = inPlacePartition(x)$;
  $inPlaceQuickSort(S, l, h-1)$;
  $inPlaceQuickSort(S, k+1, r)$;
}

In-Place Partitioning

- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).

j k
3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9 (pivot = 6)

- Repeat until j and k cross:
  - Scan j to the right until finding an element $\geq$ x.
  - Scan k to the left until finding an element < x.
  - Swap elements at indices j and k

j k
3 2 5 1 0 $\boxed{7}$ 3 5 9 $\boxed{2}$ 7 9 8 9 7 **6** 9

Summary of Sorting Algorithms

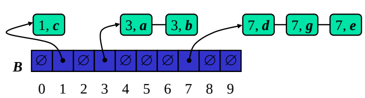
Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	in-place, randomized fastest (good for large inputs)
heap-sort	$O(n \log n)$	in-place fast (good for large inputs)
merge-sort	$O(n \log n)$	sequential data access fast (good for huge inputs)

Java Implementation

```
public static void quickSort (Object[] S, Comparator c) {
   if (S.length < 2) return; // the array is already sorted in this case
   quickSortStep(S, c, 0, S.length-1); // recursive sort method
 }
 private static void quickSortStep (Object[] S, Comparator c,
                        int leftBound, int rightBound ) {
   if (leftBound >= rightBound) return; // the indices have crossed
   Object temp;  // temp object used for swapping
   Object pivot = S[rightBound];
   int leftIndex = leftBound;     // will scan rightward
   int rightIndex = rightBound-1; // will scan leftward
   while (leftIndex <= rightIndex) { // scan right until larger than the pivot
     while ( (leftIndex <= rightIndex) && (c.compare(S[leftIndex], pivot)<=0) )
       leftIndex++;
     // scan leftward to find an element smaller than the pivot
     while ( (rightIndex >= leftIndex) && (c.compare(S[rightIndex], pivot)>=0))
       rightIndex--;
     if (leftIndex < rightIndex) { // both elements were found
       temp = S[rightIndex];
      S[rightIndex] = S[leftIndex]; // swap these elements
      S[leftIndex] = temp;
     }
   } // the loop continues until the indices cross
   temp = S[rightBound]; // swap pivot with the element at leftIndex
   S[rightBound] = S[leftIndex];
   S[leftIndex] = temp; // the pivot is now at leftIndex, so recurse
   quickSortStep(S, c, leftBound, leftIndex-1);
   quickSortStep(S, c, leftIndex+1, rightBound);
 }
```

only works for distinct elements

Bucket-Sort and Radix-Sort



Bucket-Sort

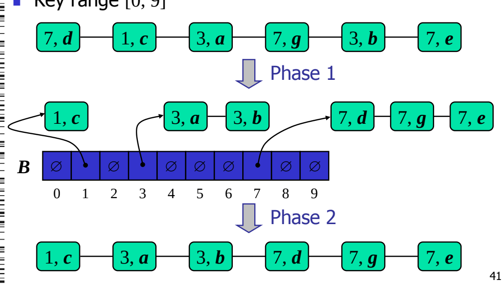
- Let be S be a sequence of n (key, element) entries with keys in the range $[0, N - 1]$
- Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)
  - Phase 1: Empty sequence S by moving each entry (k, o) into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N - 1$, move the entries of bucket $B[i]$ to the end of sequence S
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time

Bucket-sort takes $O(n + N)$ time

Algorithm *bucketSort(S, N)*
  Input sequence S of (key, element) items with keys in the range $[0, N - 1]$
  Output sequence S sorted by increasing keys
 { B = array of N empty sequences;
  while ($\neg S.isEmpty()$)
   { $f = S.first()$;
    $(k, o) = S.remove(f)$;
    $B[k].insertLast((k, o))$; }
  for ($i = 0$; i++; $i <= N - 1$)
   while ($\neg B[i].isEmpty()$)
    { $f = B[i].first()$;
     $(k, o) = B[i].remove(f)$;
     $S.insertLast((k, o))$; }
 }

Example

- Key range $[0, 9]$



Properties and Extensions

- Key-type Property
  - The keys are used as indices into an array and cannot be arbitrary objects
  - No external comparator
- **Stable** Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm

Extensions
- Integer keys in the range $[a, b]$
  - Put entry (k, o) into bucket $B[k - a]$
- String keys from a set D of possible strings, where D has constant size (e.g., names of the 50 U.S. states)
  - Sort D and compute the rank $r(k)$ of each string k of D in the sorted sequence
  - Put entry (k, o) into bucket $B[r(k)]$

Lexicographic Order

- A d-tuple is a sequence of d keys $(k_1, k_2, \ldots, k_d)$, where key k_i is said to be the i-th dimension of the tuple
- Example:
  - The Cartesian coordinates of a point in space are a 3-tuple
- The lexicographic order of two d-tuples is recursively defined as follows

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$
$$\Leftrightarrow$$
$$x_1 < y_1 \lor x_1 = y_1 \land (x_2, \ldots, x_d) < (y_2, \ldots, y_d)$$

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.

43

Lexicographic-Sort

- Let C_i be the comparator that compares two tuples by their i-th dimension
- Let ***stableSort***(S, C) be a stable sorting algorithm that uses comparator C
- Lexicographic-sort sorts a sequence of d-tuples in lexicographic order by executing d times algorithm ***stableSort***, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of ***stableSort***

Algorithm ***lexicographicSort***(S)
  Input sequence S of d-tuples
  Output sequence S sorted in lexicographic order
  { **for** ($i = d$; $i >= 1$; i--)
    stableSort(S, C_i);
  }

Example:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

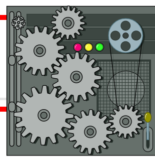
(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)

(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)

(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

44

Radix-Sort

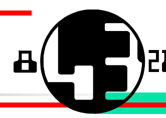


- Radix-sort is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension
- Radix-sort is applicable to tuples where the keys in each dimension i are integers in the range $[0, N-1]$
- Radix-sort runs in time $O(d(n + N))$

Algorithm ***radixSort***(S, N)
  Input sequence S of d-tuples such that $(0, \ldots, 0) \le (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \le (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in S
  Output sequence S sorted in lexicographic order
  { **for** ($i = d$; $i>=1$; i--)
    bucketSort(S, N);
  }

45

Radix-Sort for Binary Numbers

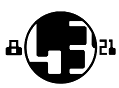


- Consider a sequence of n b-bit integers
  $x = x_{b-1} \ldots x_1 x_0$
- We represent each element as a b-tuple of integers in the range $[0, 1]$ and apply radix-sort with $N = 2$
- This application of the radix-sort algorithm runs in $O(bn)$ time
- For example, we can sort a sequence of 32-bit integers in linear time

Algorithm ***binaryRadixSort***(S)
  Input sequence S of b-bit integers
  Output sequence S sorted
  replace each element x of S with the item $(0, x)$
  { **for** ($i = 0$; $i <= b-1$; i++)
    { replace the key k of each item (k, x) of S with bit x_i of x;
      bucketSort$(S, 2)$;
    }
  }

46

Example



- Sorting a sequence of 4-bit integers

1001	0010	1001	1001	0001
0010	1110	1101	0001	0010
1101	1001	0001	0010	1001
0001	1101	0010	1101	1101
1110	0001	1110	1110	1110

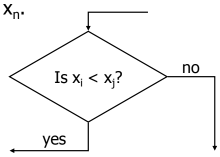
47

Sorting Lower Bound

48

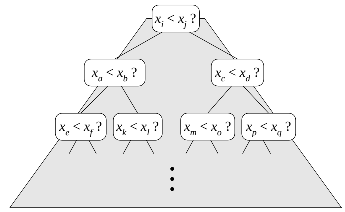
Comparison-Based Sorting

- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements, $x_1, x_2, \ldots, x_n$.



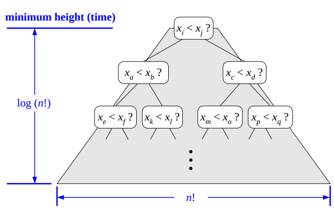
Counting Comparisons

- Let us just count comparisons then.
- Each possible run of the algorithm corresponds to a root-to-leaf path in a **decision tree**

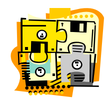


Decision Tree Height

- The height of this decision tree is a lower bound on the running time
- Every possible input permutation must lead to a separate leaf output.
  - If not, some input ...4...5... would have same output ordering as ...5...4..., which would be wrong.
- Since there are n!=1*2*...*n leaves, the height is at least log (n!)



The Lower Bound



- Any comparison-based sorting algorithms takes at least log (n!) time
- Therefore, any such algorithm takes time at least

$$\log\,(n!) \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log(n/2).$$

- That is, any comparison-based sorting algorithm must run in Ω(n log n) time.

Selection

The Selection Problem

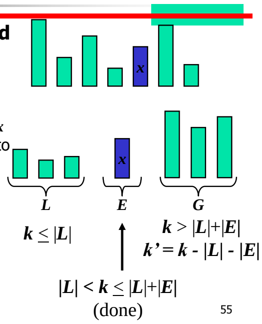
- Given an integer k and n elements $x_1, x_2, \ldots, x_n$, taken from a total order, find the k-th smallest element in this set.
- Of course, we can sort the set in O(n log n) time and then index the k-th element.

k=3 7 4 9 6 2 → 2 4 6 7 9

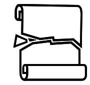
- Can we solve the selection problem faster?

Quick-Select

- Quick-select is a **randomized** selection algorithm based on the prune-and-search paradigm:
  - Prune: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Search: depending on k, either answer is in E, or we need to recurse in either L or G



Partition

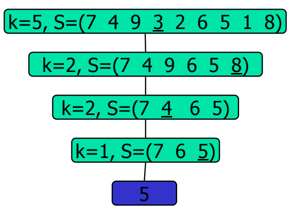


- We partition an input sequence as in the quick-sort algorithm:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-select takes $O(n)$ time

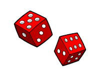
```
Algorithm partition(S, p)
  Input sequence S, position p of pivot
  Output subsequences L, E, G of the
  elements of S less than, equal to,
  or greater than the pivot, resp.
  { L, E, G = empty sequences;
    x = S.remove(p);
  while ( ¬S.isEmpty() )
    { y = S.remove(S.first());
      if ( y < x )
        L.insertLast(y);
      else if ( y = x )
        E.insertLast(y);
      else // y > x
        G.insertLast(y); }
  return L, E, G;
  }
```

Quick-Select Visualization

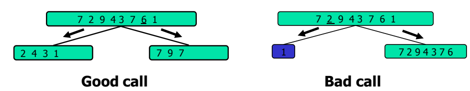
- An execution of quick-select can be visualized by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence



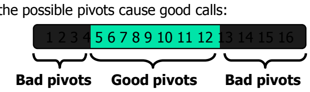
Expected Running Time (1/2)



- Consider a recursive call of quick-select on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$



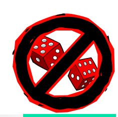
- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



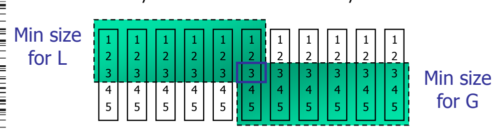
Expected Running Time (2/2)

- Probabilistic Fact #1: The expected number of coin tosses required in order to get one head is two
- Probabilistic Fact #2: Expectation is a linear function:
  - $E(X + Y) = E(X) + E(Y)$
  - $E(cX) = cE(X)$
- Let T(n) denote the expected running time of quick-select.
- By Fact #2,
  - $T(n) \leq T(3n/4) + bn*$(expected # of calls before a good call)
- By Fact #1,
  - $T(n) \leq T(3n/4) + 2bn$
- That is, T(n) is a geometric series:
  - $T(n) \leq 2bn + 2b(3/4)n + 2b(3/4)^2n + 2b(3/4)^3n + \ldots$
- So T(n) is O(n).
- We can solve the selection problem in O(n) expected time.

Deterministic Selection



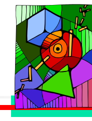
- We can do selection in O(n) worst-case time.
- Main idea: recursively use the selection algorithm itself to find a good pivot for quick-select:
  - Divide S into n/5 sets of 5 each
  - Find a median in each set
  - Recursively find the median of the "baby" medians.



- See Exercise C-11.31 for details of analysis.

Sets

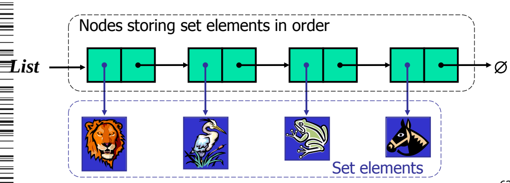
Set Operations



- We represent a set by the sorted sequence of its elements
- By specializing the auxliliary methods he generic merge algorithm can be used to perform basic set operations:
  - union
  - intersection
  - subtraction
- The running time of an operation on sets *A* and *B* should be at most $O(n_A + n_B)$
- Set union:
  - *aIsLess(a, S)*
    S.insertFirst(a)
  - *bIsLess(b, S)*
    S.insertLast(b)
  - *bothAreEqual(a, b, S)*
    S. insertLast(a)
- Set intersection:
  - *aIsLess(a, S)*
    { *do nothing* }
  - *bIsLess(b, S)*
    { *do nothing* }
  - *bothAreEqual(a, b, S)*
    S. insertLast(a)

Storing a Set in a List

- We can implement a set with a list
- Elements are stored sorted according to some canonical ordering
- The space used is $O(n)$

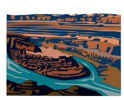


Generic Merging

- Generalized merge of two sorted lists *A* and *B*
- Template method genericMerge
- Auxiliary methods
  - aIsLess
  - bIsLess
  - bothAreEqual
- Runs in $O(n_A + n_B)$ time provided the auxiliary methods run in $O(1)$ time

```
Algorithm genericMerge(A, B)
  {  S = empty sequence;
     while ( ¬A.isEmpty() ∧ ¬B.isEmpty() )
       { a = A.first().element();  b ← B.first().element(); }
         if  ( a < b )
           { aIsLess(a, S);  A.remove(A.first()); }
         else if  ( b < a )
           { bIsLess(b, S);  B.remove(B.first()); }
         else  // b = a
           { bothAreEqual(a, b, S);
             A.remove(A.first());  B.remove(B.first()); }
     while ( ¬A.isEmpty() )
       { aIsLess(a, S);  A.remove(A.first()); }
     while ( ¬B.isEmpty() )
       { bIsLess(b, S);  B.remove(B.first()); }
     return S; }
```


Using Generic Merge for Set Operations



- Any of the set operations can be implemented using a generic merge
- For example:
  - For intersection: only copy elements that are duplicated in both list
  - For union: copy every element from both lists except for the duplicates
- All methods run in linear time.

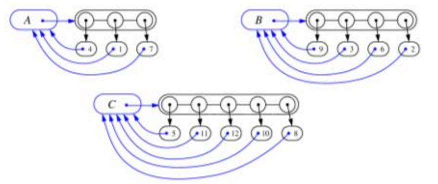
Union-Find Partition Structures

Partitions with Union-Find Operations

- makeSet(x): Create a singleton set containing the element x and return the position storing x in this set.
- union(A,B): Return the set A U B, destroying the old A and B.
- find(p): Return the set containing the element in position p.

List-based Implementation

- Each set is stored in a sequence represented with a linked-list
- Each node should store an object containing the element and a reference to the set name

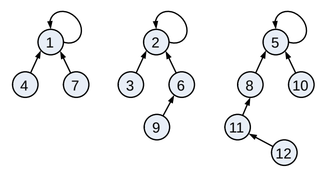


Analysis of List-based Representation

- When doing a union, always move elements from the smaller set to the larger set
  - Each time an element is moved it goes to a set of size at least double its old set
  - Thus, an element can be moved at most O(log n) times
- Total time needed to do n unions and finds is O(n log n).

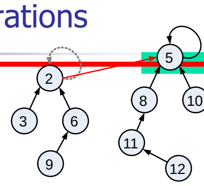
Tree-based Implementation

- Each element is stored in a node, which contains a pointer to a set name
- A node v whose set pointer points back to v is also a set name
- Each set is a tree, rooted at a node with a self-referencing set pointer
- For example: The sets "1", "2", and "5":

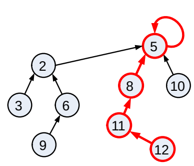


Union-Find Operations

- To do a union, simply make the root of one tree point to the root of the other

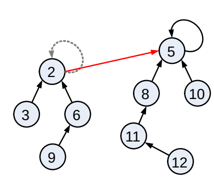


- To do a find, follow set-name pointers from the starting node until reaching a node whose set-name pointer refers back to itself



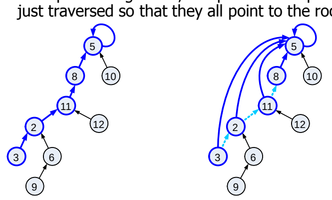
Union-Find Heuristic 1

- Union by size:
  - When performing a union, make the root of smaller tree point to the root of the larger
- Implies O(n log n) time for performing n union-find operations:
  - Each time we follow a pointer, we are going to a subtree of size at least double the size of the previous subtree
  - Thus, we will follow at most O(log n) pointers for any find.



Union-Find Heuristic 2

- Path compression:
  - After performing a find, compress all the pointers on the path just traversed so that they all point to the root



- Implies O(n log* n) time for performing n union-find operations:
  - Proof is somewhat involved… (and not in the book)

73

Proof of log* n Amortized Time

- For each node v that is a root
  - define $n(v)$ to be the size of the subtree rooted at v (including v)
  - identified a set with the root of its associated tree.
- We update the size field of v each time a set is unioned into v. Thus, if v is not a root, then $n(v)$ is the largest the subtree rooted at v can be, which occurs just before we union v into some other node whose size is at least as large as v's.
- For any node v, then, define the ***rank*** of v, which we denote as $r(v)$, as $r(v) = [\log n(v)]$:
- Thus, $n(v) \geq 2^{r(v)}$.
- Also, since there are at most n nodes in the tree of v, $r(v) = [\log n]$, for each node v.

74

Proof of log* n Amortized Time (2)

- For each node v with parent w:
  - $r(v) > r(w)$
- **Claim:** *There are at most $n/2^s$ nodes of rank s.*
- **Proof:**
  - Since $r(v) < r(w)$, for any node v with parent w, ranks are monotonically increasing as we follow parent pointers up any tree.
  - Thus, if $r(v) = r(w)$ for two nodes v and w, then the nodes counted in $n(v)$ must be separate and distinct from the nodes counted in $n(w)$.
  - If a node v is of rank s, then $n(v) \geq 2^s$.
  - Therefore, since there are at most n nodes total, there can be at most $n/2^s$ that are of rank s.

75

Proof of log* n Amortized Time (3)

- Definition: Tower of two's function:
  - $t(i) = 2^{t(i-1)}$
- Nodes v and u are in the same rank group g if
  - $g = \log^*(r(v)) = \log^*(r(u))$:
- Since the largest rank is log n, the largest rank group is
  - $\log^*(\log n) = (\log^* n)-1$

76

Proof of log* n Amortized Time (4)

- Charge 1 cyber-dollar per pointer hop during a find:
  - If w is the root or if w is in a different rank group than v, then charge the find operation one cyber-dollar.
  - Otherwise (w is not a root and v and w are in the same rank group), charge the node v one cyber-dollar.
- Since there are most $(\log^* n)-1$ rank groups, this rule guarantees that any find operation is charged at most $\log^* n$ cyber-dollars.

77

Proof of log* n Amortized Time (5)

- After we charge a node v then v will get a new parent, which is a node higher up in v's tree.
- The rank of v's new parent will be greater than the rank of v's old parent w.
- Thus, any node v can be charged at most the number of different ranks that are in v's rank group.
- If v is in rank group $g > 0$, then v can be charged at most $t(g)-t(g-1)$ times before v has a parent in a higher rank group (and from that point on, v will never be charged again). In other words, the total number, C, of cyber-dollars that can ever be charged to nodes can be bound as

$$C \leq \sum_{g=1}^{\log^* n-1} n(g)\cdot(t(g)-t(g-1))$$

78

Proof of log* n Amortized Time (end)

- Bounding $n(g)$:

$$n(g) \le \sum_{s=t(g-1)+1}^{t(g)} \frac{n}{2^s}$$

$$= \frac{n}{2^{t(g-1)+1}} \sum_{s=0}^{t(g)-t(g-1)-1} \frac{1}{2^s}$$

$$< \frac{n}{2^{t(g-1)+1}} \cdot 2$$

$$= \frac{n}{2^{t(g-1)}}$$

$$= \frac{n}{t(g)}$$

- Returning to C:

$$C < \sum_{g=1}^{\log^* n-1} \frac{n}{t(g)} \cdot (t(g) - t(g-1))$$

$$\le \sum_{g=1}^{\log^* n-1} \frac{n}{t(g)} \cdot t(g)$$

$$= \sum_{g=1}^{\log^* n-1} n$$

$$\le n \log^* n$$

79

References

1. Chapter 11, Data Structures and Algorithms by Goodrich and Tamassia.

80