CSCI2100B Data Structures
Sorting

Irwin King

king@cse.cuhk.edu.hk
http://www.cse.cuhk.edu.hk/~king

Department of Computer Science & Engineering
The Chinese University of Hong Kong

Introduction

- Sorting is simply the ordering of your data in a consistent manner, e.g., cards, telephone name list, student name list, etc.
- Each element is usually part of a collection of data called a record.
- Each record contains a key, which is the value to be sorted, and the remainder of the record consists of satellite data.
- Assumptions made here:
  - Integers
  - Use internal memory

Introduction

- There are several easy algorithms to sort in $O(n^2)$, such as insertion sort.
- There is an algorithm, **Shellsort**, that is very simple to code, runs in $o(n^2)$, and is efficient in practice.
- There are slightly more complicated O(n log n) sorting algorithms.
- Any general-purpose sorting algorithm requires Ω(n log n) comparisons.

Introduction

- Internal vs. External Sorting Methods
- Different Sorting Methods
  - Bubble Sort
  - Insertion Sort
  - Selection
  - Quick Sort
  - Heap Sort
  - Shell Sort
  - Merge Sort
  - Radix Sort

Introduction

- Types of Sorting
  - Single-pass
  - Multiple-pass
- Operations in Sorting
  - Permutation
  - Inversion (Swap)
  - Comparison

Permutation

- A permutation of a finite set S is an ordered sequence of all the elements of S, with each element appearing exactly once.
- For example, if S = {a, b, c}, there are 6 permutations of S:
- abc, acb, bac, bca, cab, cba.
- There are n! permutations of a set of n elements.

K-Permutation

- A k-permutation of S is an ordered sequence of k elements of S, with no element appearing more than once in the sequence. (Thus, an ordinary permutation is just an n-permutation of an n-set.)
- The twelve 2-permutations of the set {a, b, c, d} are
- ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc.

Inversion

- An inversion in an array of numbers is any ordered pair (i, j) having the property that i < j but a[i] > a[j].
- For example, the input list 34, 8, 64, 51, 32, 21 has nine inversions, namely (34,8), (34,32), (34,21), (64,51), (64,32), (64,21), (51,32), (51,21) and (32,21).
- Notice that this is exactly the number of swaps that needed to be (implicitly) performed by insertion sort.

Preliminaries

- Internal Sort--Each algorithm will be passed an array containing the elements and an integer containing the number of elements.
- Validity--We will assume that N, the number of elements passed to our sorting routines, has already been checked and is legal.
- Ordering--We require the existence of the "<" and ">" operators, which can be used to place a consistent ordering on the input.

Bubble Sort

- It is done by scanning the list from one-end to the other, and whenever a pair of adjacent keys is found to be out of order, then those entries are swapped.
- In this pass, the largest key in the list will have bubbled to the end, but the earlier keys may still be out of order.

Bubble Sort

- The bubble sort is probably the easiest algorithm to implement but the most time consuming of all the algorithm, other than pure random permutation.
- The basic idea underlying the bubble sort is to pass through the file sequentially several times.

Example

• Original	34	8	64	51	32	21	Number of Exchange
• ----------							
• After p = 1	8	34	21	64	51	32	4
• After p = 2	8	21	34	32	64	51	3
• After p = 3	8	21	32	34	51	64	2
• After p = 4	8	21	32	34	51	64	0
• After p = 5	8	21	32	34	51	64	0
• After p = 6	8	21	32	34	51	64	0

Bubble Sort

- Each pass consists of comparing each element in the file with its successor (x[i] with x[i+1]) and
- interchanging the two elements if they are not in proper order.
- After each pass, the largest element x[n-i] is in its proper position within the sorted array.

Bubble Sort Algorithm

- for i:= 1 to n-1 do
- for j:= n downto i+1 do
- if A[j].key < A[j-1].key then
- swap(A[j],A[j-1])

An Improved Version in C

```
bubble(x,n)
int x[], n;
{
        int hold, j, pass;
        int switched = TRUE;
        for (pass = 0; pass < n - 1 && switched == TRUE; pass++) {
                switched = FALSE;
                for (j = 0; j < n-pass-1; j++)
                        if (x[j] > x[j+1]) {
                                switched = TRUE;
                                hold = x[j];
                                x[j] = x[j+1];
                                x[j+1] = hold;
}       }       }
```


Bubble Sort

- Analysis of the bubble sort shows that there are n-1 passes and n-1 comparisons on each pass without the improvements.
- Thus the total number of comparisons is (n-1) * (n-1) = n^2 - 2n + 1, which is $O(n^2)$.
- With the first improvement the sorting will be (n-1) + (n-2) + ... + 1 = n (n+1)/2 which is also $O(n^2)$.

Bubble Sort

- The number of interchanges depends on the original order of the file.
- However, the number of interchanges cannot be greater than the number of comparisons.
- It is likely that it is the number of interchanges rather than the number of comparisons that takes up the most time in the program's execution.

Bubble Sort

- Advantage
  - It requires little additional space (one additional record to hold the temporary value for interchanging and several simple integer variables)
  - It is O(n) in the case that the file is completely sorted (or almost completely sorted since only one pass of n-1 comparisons (and no interchanges) is necessary to establish that a sorted file is sorted.

Insertion Sort

- One of the simplest sorting algorithms. It consists of n-1 passes.
- For pass p=2 through n, insertion sort ensures that the elements in positions 1 through p are in sorted order.
- Insertion sort makes use of the fact that elements in positions 1 through p-1 are already known to be in sorted order.

Insertion Example

Original	34 8 64 51 32 21	Positions Moved
---	---	---
After p = 1	34 8 64 51 32 21	0
After p = 2	8 34 64 51 32 21	1
After p = 3	8 34 64 51 32 21	0
After p = 4	8 34 51 64 32 21	1
After p = 5	8 32 34 51 64 21	3
After p = 6	8 21 32 34 51 64	4

Insertion Sort

```
insertsort(x,n)
int x[], n;
{       int i, k, y;
        for (k=1; k<n; k++) {
            y=x[k];
/* move down 1 position all elements greater than y */
            for (i = k-1; i>=0 && y < x[i]; i--)
                x[i+1] = x[i];
            /* insert y at proper position */
            x[i+1] = y;
        }
}
```


Insertion Sort

- Initially x[0] may be thought of as a sorted file of one element.
- After each repetition of the loop, the elements x[0] through x[k] are in order.

Insertion Sort Complexity

- What is the time complexity of the insertion sort algorithm when the input is sorted, in reverse order, in random order?

- The simple insertion sort may be viewed as a general selection sort in which the priority queue is implemented as an ordered array.

Insertion Sort Complexity

- Only the preprocessing phase of inserting the elements into the priority queue is necessary.
- Once the elements have been inserted, they are already sorted, so that no selection is necessary.

- If the initial file is sorted, only one comparison is made on each pass, so that the sort is O(n).
- If the file is initially sorted in the reverse order, the sort is $O(n^2)$, since the total number of comparisons is (n-1) + (n-2) + ... + 2 + 1 = (n-1) n/2 which is $O(n^2)$.

Insertion Sort Complexity

- The simple insertion sort is still usually better than the bubble sort.
  - The close the file is to sorted order, the more efficient the simple insertion sort becomes.
- The average number of comparisons in the simple insertion sort (by considering all possible permutations of the input array) is also $O(n^2)$.

Insertion Sort Complexity

- The space requirements for the sort consist of only one temporary variable, y.

- Insertion sort makes $O(n^2)$ comparisons of keys and $O(n^2)$ movements of entries. It makes $n^2 / 4 + O(n)$ comparisons of keys and movements of entries when sorting a list of length n in random order.

Insertion Sort Complexity

- Improvement: using a binary search. This reduces the total number of comparisons from $O(n^2)$ to $O(n \log n)$.
- However, the moving operation still requires $O(n^2)$ time. So the binary search does not significantly improves the overall time requirement.
- Another improvement is to use the list insertion. This reduces the time required for insertion but not the time required for searching for the proper position.

A Lower Bound for Simple Sorting

- The average number of inversions in an array of n distinct numbers is n (n - 1)/4.
  - For any list, L, of numbers, consider L_r, the list in reverse order.
  - Consider any pair of two numbers in the list (x,y), with y > x.
  - In exactly one of L and L_r this ordered pair represents an inversion.
  - The total number of these pairs is a list L and its reverse L_r is n (n-1)/2.
  - On average, it is half of the above.

A Lower Bound for Simple Sorting Algorithms

- Any algorithm that sorts by exchanging adjacent elements requires $\Omega(n^2)$ time on average.
- The average number of inversions is initially n (n-1)/4 = $\Omega(n^2)$.
- Each swap removes only one inversion, so $\Omega(n^2)$ swaps are required.

Selection Sort

- It is also called Straight Selection or Push-Down Sort.
- A selection sort is one in which successive elements are selected in order and placed into their proper sorted positions.
- The elements of the input may have to be preprocessed to make the ordered selection possible.

Selection Sort

• Original	34	8	64	51	32	21	Positions Moved
• ---	---	---	---	---	---	---	---
• After p = 1	34	8	64	51	32	21	1
• After p = 2	8	34	64	51	32	21	4
• After p = 3	8	21	64	51	32	34	2
• After p = 4	8	21	32	51	64	34	2
• After p = 5	8	21	32	34	64	51	1
• After p = 6	8	21	32	34	51	64	0

Selection Sort

- As the first stage, one scans the list to find the entry that comes last in the order.
- This entry is then interchanged with an entry in the last position.
- Now, one could repeat the process on the shorter list obtained by omitting the last entry.

Comparison

	Selection	Insertion (average)
• Assignments		
• of entries	$3.0n + O(1)$	$0.25n^2 + O(n)$
• Comparisons		
• of keys	$0.5n^2 + O(n)$	$0.25n^2 + O(n)$

Straight Selection Sort

```
selectsort(x,n)
int x[], n;
{       int i, indx, j, large;
        for (i=n-1; i>0; i--) {
            /* place the largest number of x[0] through */
        /* x[i] into large and its index into indx */
        large = x[0]; indx = 0;
        for (j=1; j <= i; j++)
            if (x[j] > large) {
                large = x[j];  indx = j;}
        x[indx] = x[i];
        x[i] = large;}}
```


Selection Sort Analysis

- The algorithm consists entirely of a selection phase in which the largest of the remaining elements, ***large***, is repeatedly placed in its proper position, i, at the end of the array.
- To do so, large is interchanged with the element x[i].
- The initial n-element priority queue is reduced by one element after each selection.

Selection Sort Analysis

- Analysis of the straight selection sort is straightforward.
- The first pass makes n-1 comparisons, the second pass makes n-2, and so on.
- Therefore there is a total of (n-1) + (n-2) + ... + 1 = (n-1) n / 2 comparisons, which is $O(n^2)$.

Selection Sort Analysis

- The number of interchanges is always n-1 (unless a test is added to prevent the interchanging of an element with itself).
- There is little additional storage required (except to hold a few temporary variables).
- The sort may be categorized as $O(n^2)$, although it is faster than the bubble sort.
- There is no improvement if the input file is completely sorted or unsorted, since the testing proceeds to completion without regard to the makeup of the file.

Shell Sort

- It is also called Diminishing Increment Sort.
- Selection sort moves the entries very efficiently but does many redundant comparisons.
- In its best case, insertion sort does the minimum number of comparisons, but is inefficient in moving entries only one place at a time.

Shell Sort

- If we were to modify the comparison method so that it first compares keys far apart, then it could sort the entries far apart.
- Afterward, the entries closer together would be sorted, and finally the increment between keys being compared would be reduced to 1, to ensure that the list is completely in order.

Example

- Original 81 94 11 96 12 35 17 95 28 58 41 75 15
- ----------
- After 5-sort 35 17 11 28 12 41 75 15 96 58 81 94 95
- After 3-sort 28 12 11 35 15 41 58 17 94 75 81 96 95
- After 1-sort 11 12 15 17 28 35 41 58 75 81 94 95 96

Shell Sort Algorithm

```
shellsort(x,n,incrmnts, numinc)
int x[], n, incrmnts[], numinc;
{       int incr, j, k, span, y;
        for (incr = 0; incr < numinc; incr++) {
            /* span is the size of the increment */
            span = incrmnts[incr];
        for (j = span; j < n; j++) {
            /* Insert element x[j] into its proper */
            /* position within its subfile */
            y = x[j];
            for (k = j-span; k >= 0 && y < x[k]; k-= span)
                x[k+span] = x[k];  x[k+span] = y;
            }}}
```


Shell Sort Analysis

- Since the first increment used by the Shell sort is large, the individual subfiles are quite small, so that the simple insertion sorts on those subfiles are fairly fast. Each sort of a subfile causes the entire file to be more nearly sorted.

- Although successive passes of the Shell sort use smaller increments and therefore deal with larger subfiles, those subfiles are almost sorted due to the actions of previous passes.

Analysis

- Thus, the insertion sorts on those subfiles are also quite efficient.
- If a file is partially sorted using an increment k and is subsequently partially sorted using an increment j, the file remains partially sorted on the increment k.
- Hence, subsequent partial sorts do not disturb earlier ones.

Analysis

- One requirement that is intuitively clear is that the elements of *incrmnts* should be relatively prime (that is, have no common divisors other than 1).
- This guarantees that successive iterations intermingle subfiles so that the entire file is indeed almost sorted when span equals 1 on the last iteration.

Analysis

- The analysis of the Shell sort is difficult at best.
- The empirical studies have been made of Shell Sort, when n is large, is in the range of $n^{1.25}$ to $1.6n^{1.25}$.
- It has been shown that the order of the Shell sort can be approximated by $O(n (\log n)^2)$ if an appropriate sequence of increments is used.
- For other series of increments, the running time can be proven to be $O(n^{1.5})$.

Analysis

- Empirical data indicates that the running time is of the form a * n^b, where a is between 1.1 and 1.7 and b is approximately 1.26, or of the form c * n $(\log n)^2$ - d * n * log n, where c is approximately 0.3 and d is between 1.2 and 1.75.
- In general the Shell sort is recommended for moderately sized files of several hundred elements.

Analysis

- Knuth recommends choosing increments as follows:
  - Define a function h recursively so that h(1) = 1 and h(i+1) = 3 * h(i) + 1.
- Let x be the smallest integer such that h(x) ≥ n, and set *numinc*, the number of increments, to x-2 and *incrmnts*[i] to h(*numinc* -i +1) for i from 1 to *numinc*.

Heapsort

- Priority queues can be used to sort in O(n log n) time.
- The algorithm based on this idea is known as **heapsort** and gives the best Big-Oh running time we have seen so far.

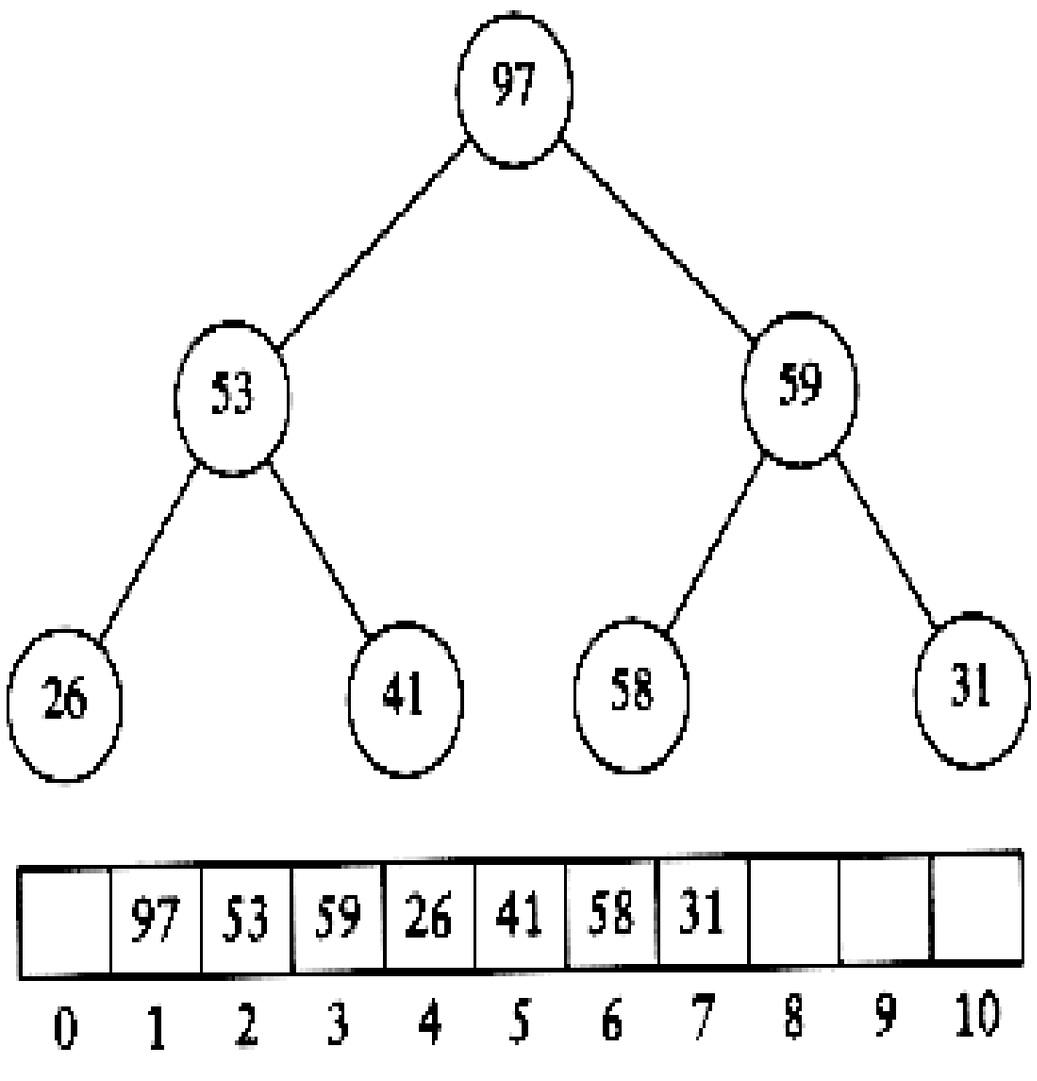
Basic Idea

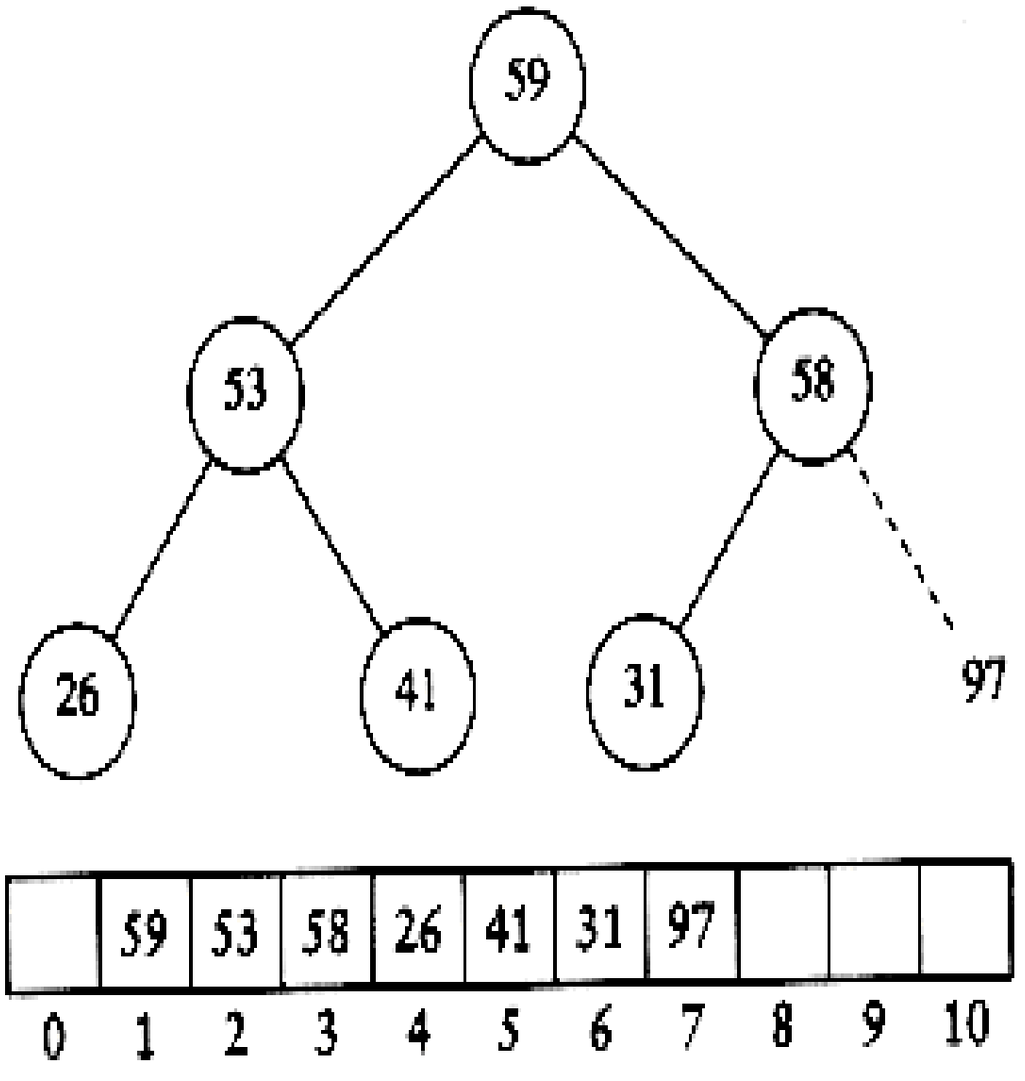
- Build a binary heap of n elements. This stage takes O(n) time.
- We then perform n ***delete_min*** operations.
- The elements leave the heap smallest first, in sorted order.
- By recording these elements in a second array and then copying the array back, we sort n elements.
- Since each ***delete_min*** takes O(log n) time, the total running time is O(n log n).

Cleaver Modification

- The main problem with this algorithm is that it uses an extra array. Thus, the memory requirement is doubled.
- Use the last cell
- A clever way to avoid using a second array makes use of the fact that after each ***delete_min***, the heap shrinks by 1.
- Thus the cell that was last in the heap can be used to store the element that was just deleted.

Example

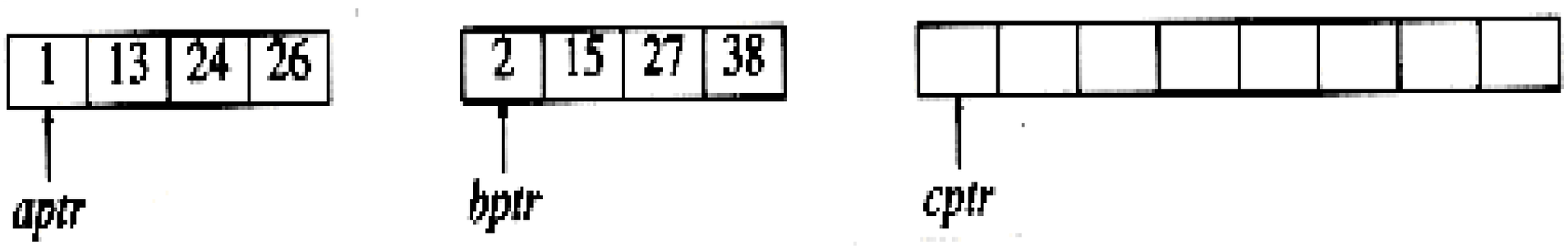


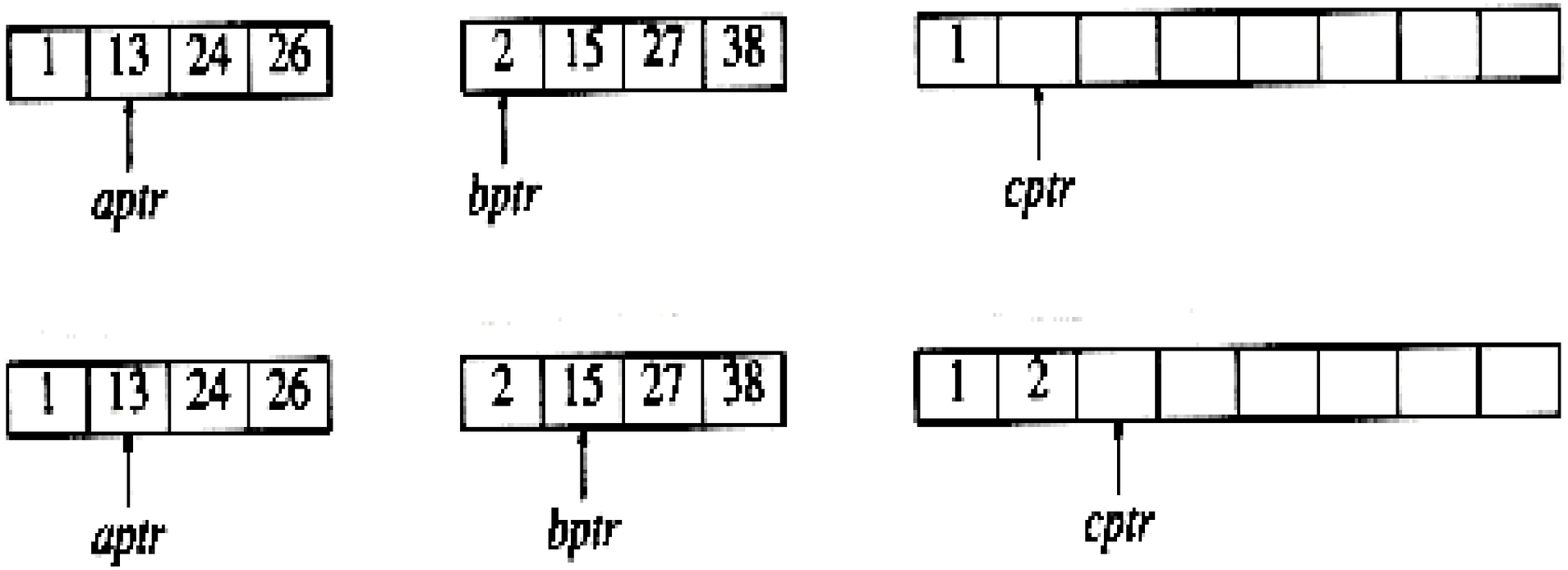


Merge Sort

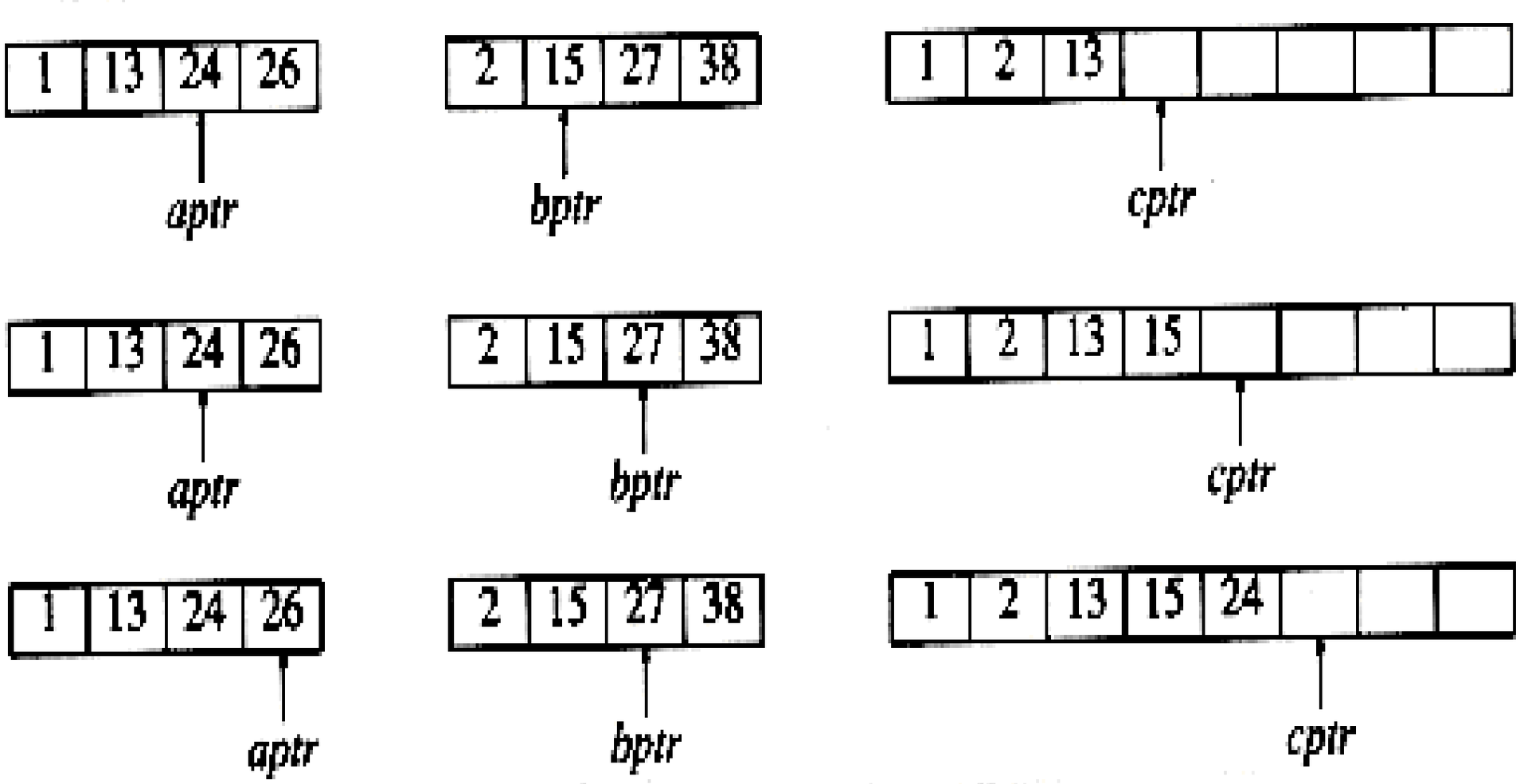
- Mergesort is an excellent method for external sorting, that is, for problems in which the data are kept on disks or magnetic tapes, not in high-speed memory.

Merge Sort

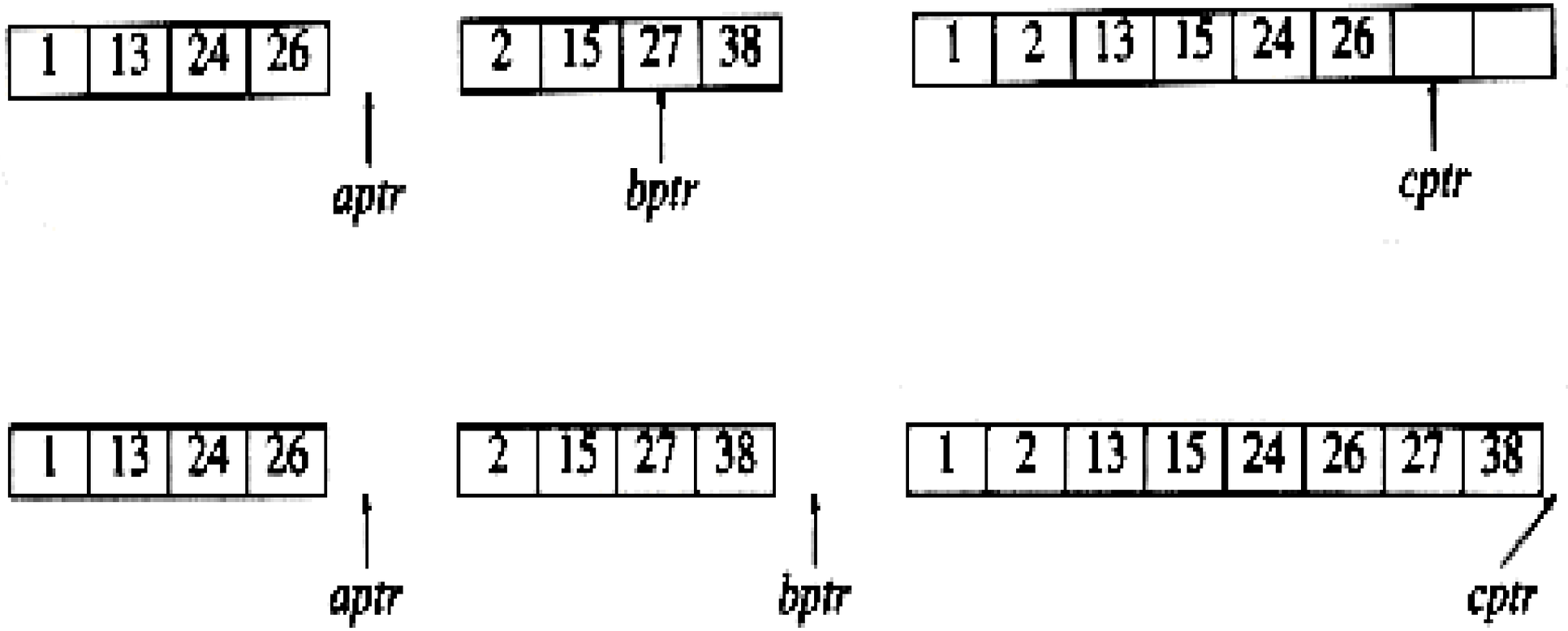




Merge Sort



Merge Sort



Merge Sort

- This algorithm is a classic divide-and-conquer strategy.
- The problem is divided into smaller problems and solved recursively.
- The conquering phase consists of patching together the answers.
- Divide-and-conquer is a very powerful use of recursion that we will see many times.

Merge Sort Analysis

- Comparison of keys is done at only one place in the complete mergesort procedure.
- This place is within the main loop of the merge procedure.
- After each comparison, one of the two nodes is sent to the output list.
- Hence the number of comparisons certainly cannot exceed the number of nodes being merged.

Analysis

- It is clear from the tree that the total lengths of the lists on each level is precisely n, the total number of entries.
- In other words, every entry is treated in precisely one merge on each level.
- Hence the total number of comparisons done on each level cannot exceed n.
- The number of levels, excluding the leaves (for which no merges are done), is log n rounded up to the next smallest integer.
- The number of comparisons of keys done by mergesort on a list of n entries, therefore, is no more than n log n rounded upward.

Analysis

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n$$

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$$

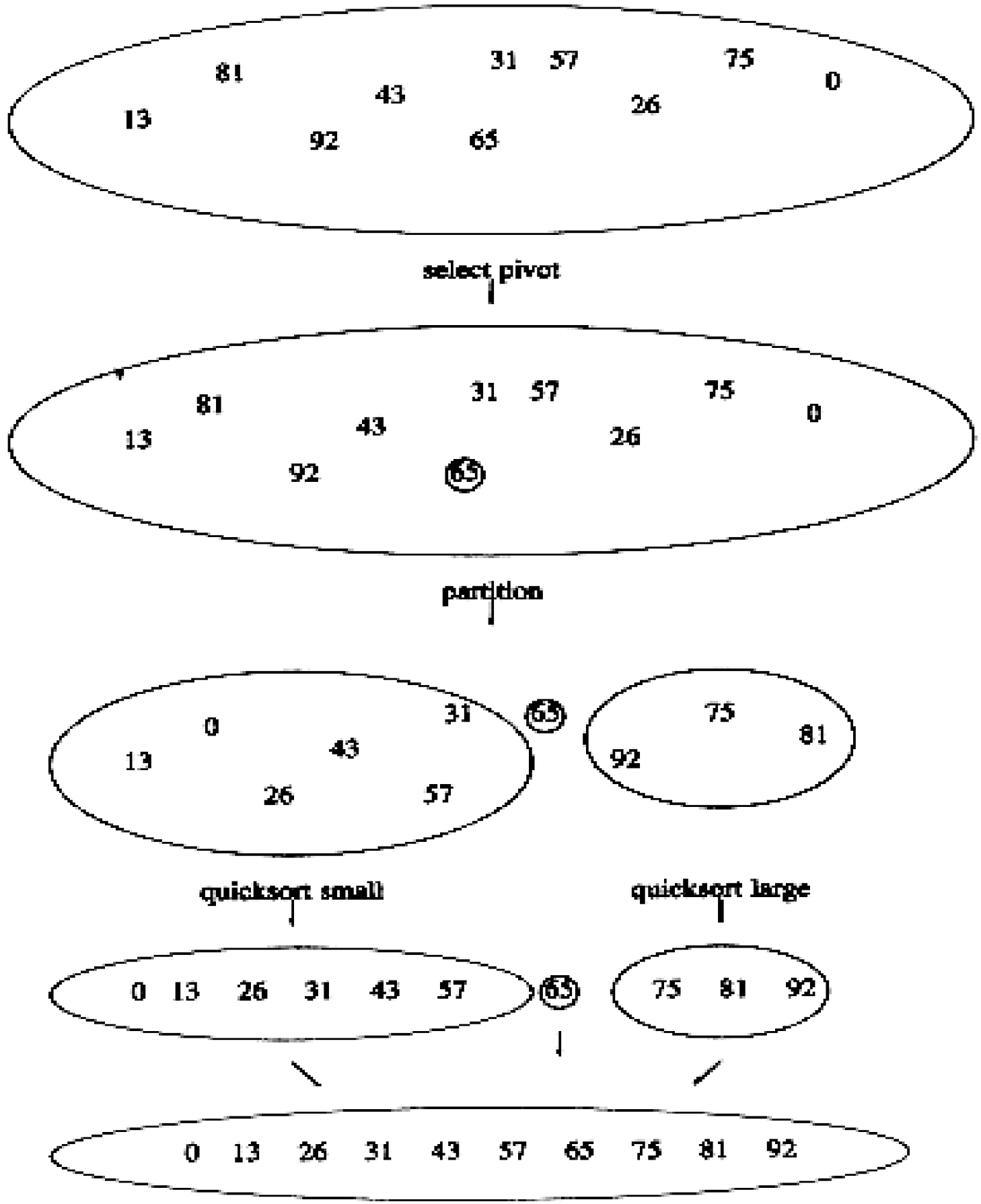
M

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log n$$

$$T(n) = n\log n + n = O(n\log n)$$

QuickSort


81
31
57
75
0
43
26
13
92
65
select pivot
81
31
57
75
0
43
13
26
92
65
partition
0
31
65
75
81
43
13
92
26
57
quicksort small
quicksort large
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

Quicksort

- Divide-and-conquer process for sorting a typical subarray A[p . . r]
- Divide: The array A[p . . r] is partitioned (rearranged) into two nonempty subarrays A[p . . q] and A[q + 1 . . r] such that each element of A[p . . q] is less than or equal to each element of A[q + 1 . . r].
- The index q is computed as part of this partitioning procedure.

Quicksort

- Conquer: The two subarrays A[p . . q] and A[q + 1 . . r] are sorted by recursive calls to quicksort.

- Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire array A[p . . r] is now sorted.

Example

R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	m	n
[26	5	37	1	61	11	59	15	48	19	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

Quicksort Algorithm

- QUICKSORT(A,p,r)
- 1 if p < r
- 2 then q PARTITION(A,p,r)
- 3 QUICKSORT(A,p,q)
- 4 QUICKSORT(A,q + 1,r)

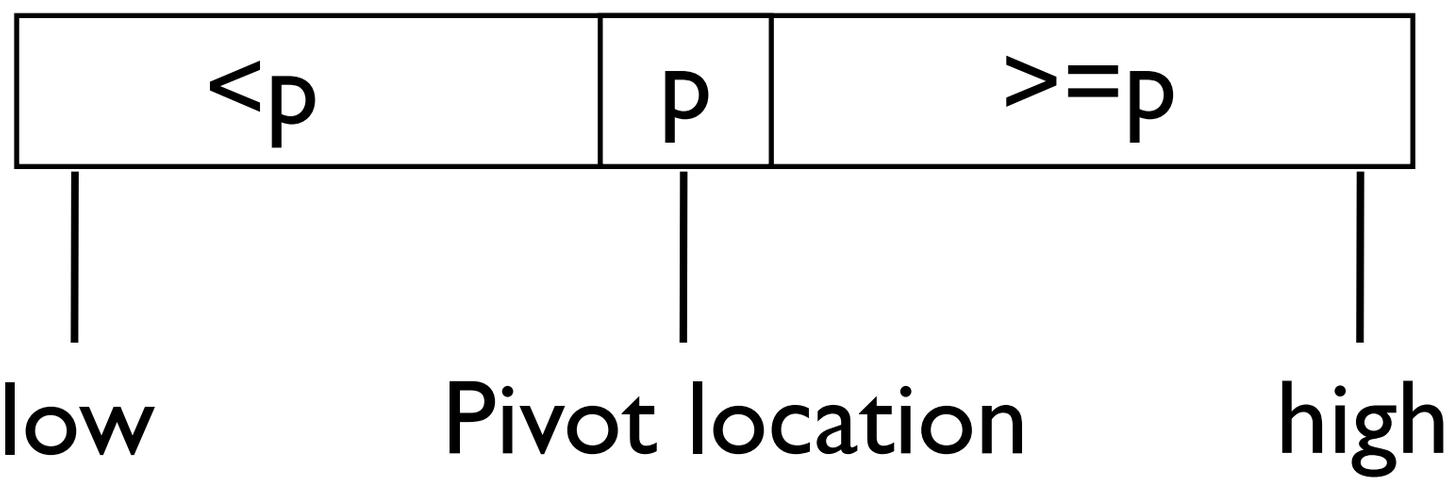
- To sort an entire array A, the initial call is QUICKSORT(A, 1, length[A]).

Partition Algorithm

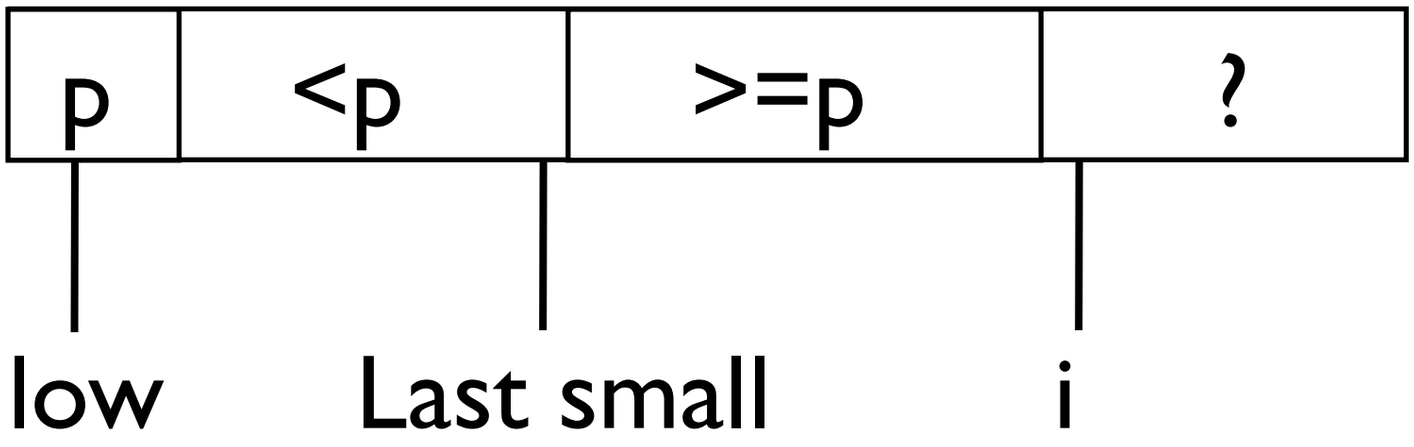
```
partition(x,lb,ub,pj)
int x[], lb, ub, *pj;
{       int a, down, temp, up;
        a = x[lb]; up = ub; down = lb;
        while (down < up) {
            while (x[down] <= a && down < ub)
                down++;
            while (x[up] > a)    up--;
            if (down < up) {
                temp = x[down];   x[down] = x[up];
                x[up] = temp;}}
    x[lb] = x[up]; x[up] = a; *pj = up;}
```


Partition Operation

- Goal

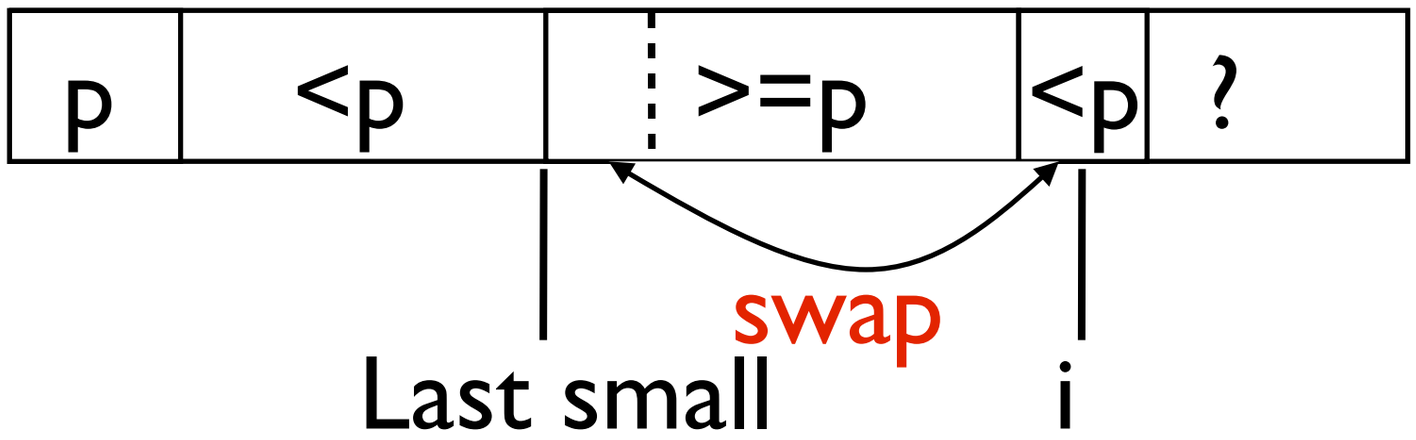


- Loop Invariant

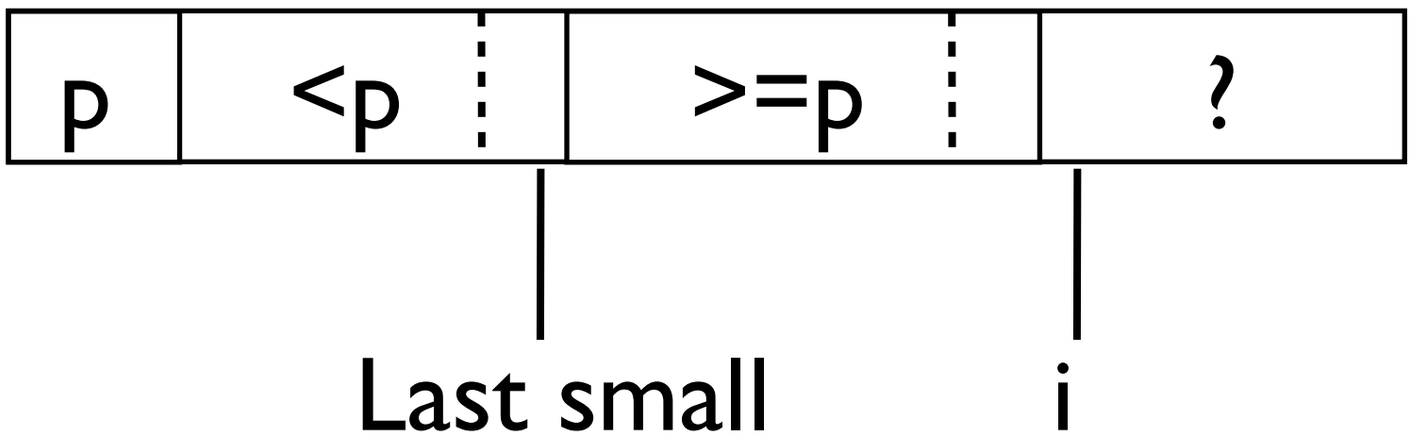


Partition Operation

- Case 1- new element is <p



- Case 2- New element is >= p



Quick Sort Analysis

- The number of comparisons of keys that will have been made in the call to partition is n-1, since every entry in the list is compared to the pivot, except for the pivot entry itself.

Analysis

- Let us denote by C(n) the average number of comparisons done by quicksort on a list of length n and by C(n,p) the average number of comparisons on a list of length n where the pivot for the first partition turns out to be p.
- The remaining work is then C(p-1) and C(n-p) comparisons for the sublists.
- So for $n \geq 2$, we have
- $C(n,p) = (n-1) + C(p-1) + C(n-p)$.

Analysis

- The average time is then
- $C(n) = (n-1) + 2/n (C(0) + C(1) + \ldots + C(n-1))$.
- For a list of length n-1, we have
- $C(n-1) = (n-2) + 2/n -1 (C(0) + C(1) + ... + C(n-2))$.
- Multiplying the first expression by n, the second by n-1, and subtracting, we obtain
- $nC(n)-(n-1) C(n-1) = n(n-1)-(n-1)(n-2)+2C(n-1)$,

Analysis

- which can be rearranged as
- C(n)-2/(n+1) = C(n-1)-2/n + 2/(n+1)

- C(n)-2/(n+1) = (C(n-1)-2)/n + 2/(n+1)
- = (C(n-2)-2)/(n-1) + 2/(n+1) + 2/n
- = (C(n-3)-2)/(n-2) + 2/(n+1) + 2/n + 2/(n-1)
- = (C(2)-2)/3 + 2/(n+1) + 2/n + 2/(n -1) + ... + 2/4
- = (C(1)-2)/2 + 2/(n +1) + 2/n + 2/(n-1) + ... + 2/4 + 2/3...
- = -1 + 2(1/(n+1) + 1/n + 1/(n-1) + ...+ 1/4 + 1/3).

Harmonic Numbers

- H(n) = 1 + 1/2 + ... + 1/n
- Evaluating the above Harmonic number we find that

$$\int_{\frac{1}{2}}^{n+\frac{1}{2}} \frac{1}{x}\,dx = \log(n+\frac{1}{2}) - \log\frac{1}{2} \approx \log n + 0.7$$

- This shows that
- $H(n) = 1 + 1/2 + \ldots + 1/n = \log n + O(1)$.
- Substitute this back to the original equation and we obtain
- $(C(n)-2)/(n+1) = 2 \log n + O(1)$.

Harmonic Numbers

- The final step is to solve this equation for C(n), noting that, when we multiply O(1) by n+1, we obtain an expression that is O(n).

- In its average case, quicksort performs
- $$C(n) = 2n \log n + O(n)$$
- comparisons of keys in sorting a list of n entries.

Observations on Quick Sort- Choice of Pivot

- We can choose any entry we wish and swap it with the first entry before beginning the loop that partitions the list.
- Often, the first entry is a poor choice since if the list is already sorted, then the first key will have no others less than it, and so one of the sublists will be empty.
- Hence, it might be better to select an entry near the center of the list in hope that the entry will partition the keys so that about half come on each side of the pivot.

Bucket/Radix/Postman Sort

- In some special cases, the sorting can be performed in linear time.
- The idea is to consider the key one character at a time and to divide the items, not into two sublists, but into as many sublists as there are possibilities for the given character from the key.
- If our keys, for example, are words or other alphabetic strings, then we divide the list into 26 sublists at each stage.
- That is, we set up a ***table*** of 26 lists and distribute the items into the lists according to one of the characters in the key.

Radix Sort Analysis

- The time used by radix sort is proportional to nk, where n is the number of items being sorted and k is the number of characters in a key.
- The time for all our other sorting methods depends on n but not directly on the length of a key.
- The best time was that of mergesort, which was n log n + O(n).

Analysis

- The relative performance of the methods will therefore relate in some ways to the relative sizes of nk and n log n.
- If the keys are long but there are relatively few of them, then k is large and n relatively small, and other methods (such as mergesort) will outperform radix sort.
- But if k is small and there are a large number of keys, then radix sort will be faster than any other method we have studied.

Other Sorting Issues-Sorting Large Structures

- Problem: When sorting large structures sometimes it is impractical to store all the information of a record in an array since it is expensive and swapping of large records are inefficient.
- If this is the case, a practical solution is to have the input array contain pointers to the structures.

Sorting Large Structures

- We sort by comparing the keys the pointers point to, swapping pointers when necessary.
  - All the data movement is essentially the same as if we were sorting integers.
- This is known as indirect sorting.
- We can use this technique for most of the data structures we have described.
- This is sometimes called sorting by address.

Stability

- A sorting function is called stable if, whenever two items have equal keys, then on completion of the sorting function the two items will be in the same order in the list as before sorting.
- Stability is important if a list has already been sorted by one key and is now being sorted by another key, and it is desired to keep as much of the original ordering as the new one allows.

Stability

- An algorithm is stable if for all records i and j such that k[i] equals k[j], if r[i] precedes r[j] in the original file, r[i] precedes r[j] in the sorted file.
- That is a stable sort keeps records with the same key in the same relative order that they were in before the sort.

Example

- Given (a,1), (b,2), (c,3), (a,4), (a,5), (b,6), (c,7).

- The output of (a,1), (a,4), (a,5), (b,2), (b,6), (c,3), (c,7) is stable.
- Insertion sort (before equal element): (a,5), (a,4), (a,1), (b, 6), (b,2), (c,7), (c,3) is unstable.