DATA STRUCTURE
SEARCHING & SORTING
UNIT III

Learning Objectives

- Searching Techniques
- Internal Sorting Techniques
- External Sorting Techniques

INTERNAL SORTING TECHNIQUES

Learning Objectives

- Sorting Techniques & Algorithm Analysis
  - Exchange sort
  - Selection sort
  - Insertion sort
  - Shellsort
  - Mergesort
  - Quicksort
  - Heap Sort
  - Radixsort,

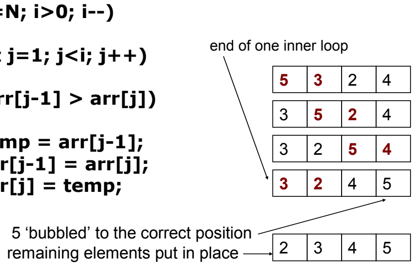
Sorting

- The objective is to take an unordered set of comparable data items and arrange them in order.
- We will usually sort the data into ascending order — sorting into descending order is very similar.
- Data can be sorted in various ADTs, such as arrays and trees.

Bubble sort

A pretty dreadful type of sort! However, the code is small:

```
for (int i=N; i>0; i--)
{
   for (int j=1; j<i; j++)
   {
      if (arr[j-1] > arr[j])
      {
         temp = arr[j-1];
         arr[j-1] = arr[j];
         arr[j] = temp;
      }
   }
}
```



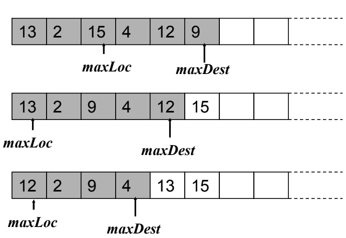
Selection Sort

- Find the largest element in the array [0:N-1], place it at the location N-1
- Find the largest element in the array [0:N-2], place it at the location N-2
- And so on…
- The major disadvantage is the performance overhead of finding the largest element at each step

Selection Sort: algorithm

- Initialise ***maxDest*** to the last index of the Array
- Search from the start of the array to ***maxDest*** for the largest element: call its position ***maxLoc***
- Swap element indexed by ***maxLoc*** with element indexed by ***maxDest***
- Decrement ***maxDest*** by one
- Repeat steps 2 – 4

Selection Sort



etc.

Time complexity of Selection Sort

The main operations being performed in the algorithm are:

1. Comparisons to find the largest element in the subarray (to find ***maxLoc*** index): there are '$n + (n-1) + \ldots + 2 + 1$' comparisons, i.e., $n/2\ (n+1) = (n^2 + n) / 2$ so this is an $O(n^2)$ operation
2. Swapping elements between ***maxLoc*** and ***maxDest***: n-1 exchanges are performed so this is an O(n) operation

- The dominant operation (time-wise) gives the overall time complexity, i.e., $O(n^2)$
- Although this is an $O(n^2)$ algorithm, its advantage over $O(n \log n)$ sorts is its simplicity

Time complexity of Selection Sort

- For very small sets of data, SelectionSort may actually be more efficient than $O(n \log n)$ algorithms
- This is because many of the more complex sorts have a relatively high level of overhead associated with them, e.g., recursion is expensive compared with simple loop iteration
- This overhead might outweigh the gains provided by a more complex algorithm where a small number of data elements is being sorted
- SelectionSort does better than BubbleSort as fewer swaps are required, although the same number of comparison operations are performed (each swap puts an element in its correct place)

Insertion sort

Tree Insertion Sort

- This is inserting into a normal tree structure:
- i.e. data are put into the correct position when they are inserted.
- Requires a find and an insert.
- The time complexity for one insert is O(*logN*) + O(1) = O(*logN*);
- therefore to insert *N* items will have a complexity of O(*NlogN*).

Insertion sort

Array Insertion Sort

The array must be sorted; insertion requires a find + insert

.

To insert *N* items will have a complexity of **O(*N*²)**.

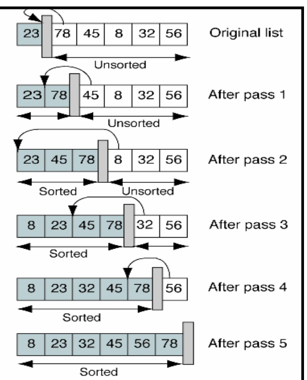


FIGURE 12-8 Insertion Sort Example

Array Insertion Sort

10	5	4	7	3	*Temp=5*
10	10	4	7	3	*A[0]= temp*

Array Insertion Sort

5	10	4	7	3

Temp=4

5	10	10	7	3

5	5	10	7	3

A[0]= temp

4	5	10	7	3

Array Insertion Sort

4	5	10	7	3

Temp=7

4	5	10	10	3

A[2]= temp

4	5	7	10	3

Temp=3

4	5	7	10	10

Array Insertion Sort

4	5	7	7	10

4	5	5	7	10

4	4	5	7	10

A[0]= temp

3	4	5	7	10

Array Insertion Sort: ALGO

```
FOR (P=1; P<N; P++)
   BEGIN
        temp= A[P];
        FOR (j=P; j>0 && A[j-1]>temp; j--)
                A[j]= A[j-1]
        A[j]= temp;
   END
END
```


Shell sort [diminishing increment sorting]

Named after D.L. Shell! But it is also rather like shrinking shells of sorting, for Insertion Sort.

Shell sort aims to reduce the work done by insertion sort (i.e. scanning a list and inserting into the right position).

Do the following:

- Begin by looking at the lists of elements x_1 elements apart and sort those elements by insertion sort
- Reduce the number x_1 to x_2

Shell Sort: Illustration

Gap=3

10	3	5	21	7	6	4	2	43	54

10	3	5	21	7	6	4	2	43	54

10	3	5	21	7	6	4	2	43	54

10	3	5	21	7	6	4	2	43	54

4	3	5	10	7	6	21	2	43	54

Shell Sort: Illustration

Gap =3

4	3	5	10	7	6	21	2	43	54

4	2	5	10	3	6	21	7	43	54

4	2	5	10	3	6	21	7	43	54

Continue for other values of *gap*…

Shell Sort: Algorithm

```
For Each Gap in GapValues
   For I=Gap to N-1
        Temp= A[I]
        For (J=I; J>=Gap && A[J-Gap]>Temp; J=J-Gap)
                A[J]= A[J-Gap]
        End For
        A[J]= Temp
   End For
End Form
```


How do you choose the gap size?

- The idea of the decreasing gap size is that the list becomes more and more sorted each time the gap size is reduced,
- Therefore (for example) having a gap size of 4 followed by a gap size of 2 is not a good idea, because you'll be sorting half the numbers a second time.
- There is no formal proof of a good initial gap size, but about a 10^{th} the size of N is considered to be a reasonable start.
- Try to use prime numbers as gap size, or odd numbers *if a list of primes is not feasible to generate* (though note gaps of 9, 7, 5, 3, 1 will be doing less work when gap=3).

Shell Sort

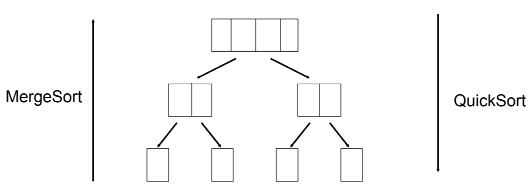
Running time of Shell sort depends upon the gap sequence chosen.

The set of gap values suggested by Shell, (N/2, N/4, …, 1) give a worst case running time of $O(N^2)$

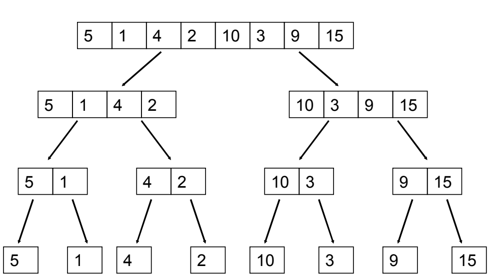
Consider set

$\mathbf{1}_{\mathbf{0}}$, 9_{1}, $\mathbf{2}_{\mathbf{2}}$, 10_{3}, $\mathbf{3}_{\mathbf{4}}$, 11_{5}, $\mathbf{4}_{\mathbf{6}}$, 12_{7},
$\mathbf{5}_{\mathbf{8}}$, 13_{9}, $\mathbf{6}_{\mathbf{10}}$, 14_{11}, $\mathbf{7}_{\mathbf{12}}$, 15_{13}, $\mathbf{8}_{\mathbf{14}}$, 16_{15}

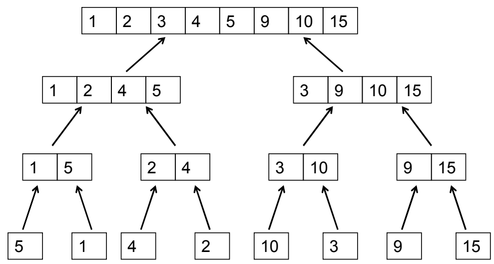
Divide and conquer sorting



Divide ...



and conquer

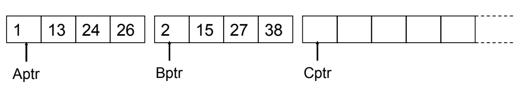


Merge Sort

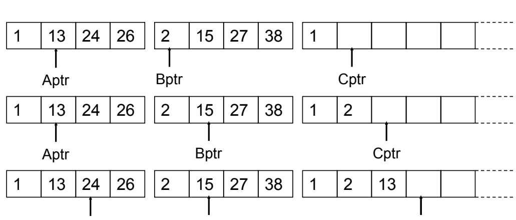
For MergeSort an initial array is repeatedly divided into halves, until arrays of just one or zero elements remain.

At each level of recombination, two sorted arrays are merged into one.

This is done by copying the smaller of the two elements from the sorted arrays into the new array, and then moving along the arrays.



Merging



etc.

Analysis of MergeSort

Let the time to carry out a MergeSort on n elements be $T(n)$

Assume that n is a power of 2, so that we always split into equal halves (the analysis can be refined for the general case)

For $n=1$, the time is constant, so we can take $T(1) = 1$

Otherwise, the time $T(n)$ is the time to do two MergeSorts on $n/2$ elements, plus the time to merge, which is linear

So, $T(n) = 2\ T(n/2) + n$

Analysis of MergeSort cont..

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

Divide through by n to get

- $T(n)/n = T(n/2)/(n/2) + 1$

Replacing n by $n/2$ gives,

- $T(n/2)/(n/2) = T(n/4)/(n/4) + 1$

And again gives,

- $T(n/4)/(n/4) = T(n/8)/(n/8) + 1$

We continue until we end up with

- $T(2)/2 = T(1)/1 + 1$

Analysis of MergeSort (2)

Since n is divided by 2 at each step, we have $\log_2 n$ steps

Now, substituting the last equation in the previous one, and working back up to the top gives

- $T(n)/n = T(1)/1 + \log_2 n$
- $T(n)/n = \log_2 n + 1$

So $T(n) = n \log_2 n + n = O(n \log n)$

Although this is an **$O(n \log n)$** algorithm, it is hardly ever used for main memory sorts because it requires linear extra memory

Merge Sort

```
MergeSort(A, tmpA, left, right) {
  if (left < right) {
      mid = floor((left + right) / 2);
      MergeSort(A, tmpA, left, mid);
      MergeSort(A, tmpA, mid+1, right);
      Merge(A, tmpA, left, mid+1, right);
  }
}
// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A.
// It requires O(n)
// time, and *does* require allocating O(n) space
```


Merge Sort: Analysis

```
Statement                                    Effort
MergeSort(A,tmpA, left, right) {             T(n)
   if (left < right) {                       O(1)
      mid = floor((left + right) / 2);       O(1)
      MergeSort(A,tmpA, left, mid);          T(n/2)
      MergeSort(A, tmpA, mid+1, right);      T(n/2)
      Merge(A, tmpA, left, mid+1, right);    O(n)
   }
}
```

So T(n) = O(1) when n = 1, and
2T(n/2) + O(n) when n > 1

This expression is a *recurrence*

Merge Function

```
void Merge(A, tmpA,  Lpos, Rpos, Rend)
{
Lend = Rpos-1;
TmpPos=Lpos;
NoElements= Rend-Lpos+1;
While(Lpos<=Lend && Rpos <=Rend)
  If(A[Lpos] <= A[Rpos])
       tmpA[TmpPos++] = A[Lpos++];
  else
       tmpA[TmpPos++] = A[Rpos++];
```

Merge Function Contd..

```
    While(Lpos <= Lend)
      tmpA[TmpPos++] = A[Lpos++];

    While(Rpos <= Rend)
      tmpA[TmpPos++] = A[Rpos++];

    For(i=0; i<NoElements; i++, Rend--)
      A[Rend]=tempA[Rend]
}
```


QuickSort

As its name implies, QuickSort is the fastest known sorting algorithm *in practice*

It was devised by C.A.R. Hoare in 1962

Its average running time is **O(*n* log *n*)** and it is very fast

It has worst-case performance of **O(n^2)** but this can be made very unlikely with little effort

QuickSort

The idea is as follows:

1. **If the number of elements to be sorted is 0 or 1, then return**
2. **Pick any element, *v* (this is called the *pivot*)**
3. **Partition the other elements into two disjoint sets, S_1 of elements $\leq v$, and S_2 of elements $> v$**
4. **Return QuickSort (S_1) followed by *v* followed by QuickSort (S_2)**

Review: Quicksort

Another divide-and-conquer algorithm

- **The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]**
  - ✓**Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]**
- **The subarrays are recursively sorted by calls to quicksort**
- **Unlike merge sort, no combining step: two subarrays form an already-sorted array**

Review: Partition

Clearly, all the action takes place in the `partition()` function

- Rearranges the subarray **in place**
- End result:
  - ✓Two subarrays
  - ✓All values in first subarray ≤ all values in second
- Returns the index of the "pivot" element separating the two subarrays

QuickSort example

5	1	4	2	10	3	9	15	12

Pick the middle element as the pivot, i.e., 10

Partition into the two subsets below

5	1	4	2	3	9

15	12

Sort the subsets

1	2	3	4	5	9

12	15

Recombine with the pivot

1	2	3	4	5	9	10	12	15

Partitioning example

5	11	4	25	10	3	9	15	12

Pick the middle element as the pivot, i.e., 10

Move the pivot out of the way by swapping it with the first element

10	11	**4**	25	5	3	9	15	12

swapPos

Step along the array, swapping small elements into swapPos

10	**4**	**11**	25	**5**	3	9	15	12

swapPos

Partitioning example (2)

10	4	**5**	25	**11**	**3**	9	15	12

swapPos

10	4	5	**3**	11	**25**	9	15	12

swapPos

10	4	5	3	**9**	25	11	15	12

swapPos

9	4	5	3	**10**	25	11	15	12

Partition

Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

Pseudo code for partitioning

```
pivotPos = middle of array a;
swap a[pivotPos] with a[first];   // Move the pivot out of the way
swapPos = first + 1;
for each element in the array from swapPos to last do:
   // If the current element is smaller than pivot we
   // move it towards start of array
   if (a[currentElement] < a[first]):
      swap a[swapPos] with a[currentElement];
       increment swapPos by 1;
// Now move the pivot back to its rightful place
swap a[first] with a[swapPos-1];
return swapPos-1;   // Pivot position
```


Some observations about QuickSort

- A consistently poor choice of pivot can lead to $O(n^2)$ time performance
- A good strategy is to pick the middle value of the left, centre, and right elements
- For small arrays, with n less than (say) 20, QuickSort does not perform as well as simpler sorts such as SelectionSort
- Because QuickSort is recursive, these small cases will occur frequently
- A common solution is to stop the recursion at $n = 10$, say, and use a different, non-recursive sort
- This also avoids nasty special cases, e.g., trying to take the middle of three elements when n is one or two

Analysis of QuickSort

- **We assume a random choice of pivot**
- **Let the time to carry out a QuickSort on n elements be $T(n)$**
- **We have $T(0) = T(1) = 1$**
- **The running time of QuickSort is the running time of the partitioning (linear in n) plus the running time of the two recursive calls of QuickSort**
- **Let i be the number of elements in the left partition, then**
  $T(n) = T(i) + T(n-i-1) + cn$ (for some constant c)

Worst-case analysis

If the pivot is always the smallest element, then $i = 0$ always

We ignore the term $\boldsymbol{T(0) = 1}$, so the recurrence relation is

$\boldsymbol{T(n) = T(n-1) + cn}$

So, $\boldsymbol{T(n-1) = T(n-2) + c(n-1)}$ and so on until we get

$\boldsymbol{T(2) = T(1) + c(2)}$

Substituting back up gives

$\boldsymbol{T(n) = T(1) + c(n + \ldots + 2) = O(n^2)}$

Notice that this case happens if we always take the pivot to be the first element in the array and the array is already sorted

So, in this extreme case, QuickSort takes $O(n^2)$ time to do absolutely nothing!

Best-case analysis

- **In the best case, the pivot is in the middle**
- **To simplify the equations, we assume that the two subarrays are each exactly half the length of the original (a slight overestimate which is acceptable for big-Oh calculations)**
- **So, we get** $\boldsymbol{T(n) = 2T(n/2) + cn}$
- **This is very similar to the formula for MergeSort, and a similar analysis leads to**

  $\boldsymbol{T(n) = cn \log_2 n + n = O(n \log n)}$

Average-case analysis

- We assume that each of the sizes of the left partition are equally likely, and hence have probability $1/n$
- With this assumption, the average value of $T(i)$, and hence also of $\boldsymbol{T(n-i-1)}$, **is** $\boldsymbol{(T(0) + T(1) + \ldots + T(n-1))/n}$
- Hence, our recurrence relation becomes
  $\boldsymbol{T(n) = 2(T(0) + T(1) + \ldots + T(n-1))/n + cn}$
- Multiplying by n gives
  $\boldsymbol{nT(n) = 2(T(0) + T(1) + \ldots + T(n-1)) + cn^2}$
- Replacing n by $n-1$ gives
  $\boldsymbol{(n-1)T(n-1) = 2(T(0) + T(1) + \ldots + T(n-2)) + c(n-1)^2}$
- Subtracting the last equation from the previous one gives
  $\boldsymbol{nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c}$

Average-case analysis (2)

Rearranging, and dropping the insignificant c on the end, gives $nT(n) = (n+1)T(n-1) + 2cn$

Divide through by $n(n+1)$ to get $T(n)/(n+1) = T(n-1)/n + 2c/(n+1)$

Hence, $T(n-1)/n = T(n-2)/(n-1) + 2c/n$ and so on down to $T(2)/3 = T(1)/2 + 2c/3$

Substituting back up gives $T(n)/(n+1) = T(1)/2 + 2c(1/3 + 1/4 + \ldots + 1/(n+1))$

$$T(n)/(n+1) = T(1)/2 + 2c \sum_{i=3}^{n+1} [1/i]$$

Average-case analysis (2)

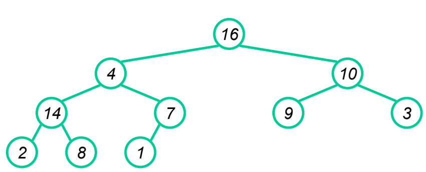
$$T(n)/(n+1) = T(1)/2 + 2c \sum_{i=3}^{n+1} [1/i]$$

The sum in brackets is about $\log_e(n+1) + \gamma - 3/2$, where γ is Euler's constant, which is approximately **0.577**

So, $T(n)/(n+1) = O(\log n)$ and $T(n) = O(n \log n)$

Review: Heaps

A *heap* is a "complete" binary tree, usually represented as an array:



A =

16	14	10	8	7	9	3	2	4	1

Review: Heaps

To represent a heap as an array:

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```


Review: The Heap Property

Heaps also satisfy the *heap property*:

A[*Parent*(*i*)] ≥ A[*i*] for all nodes *i* > 1

- In other words, the value of a node is at most the value of its parent
- The largest value is thus stored at the root (A[1])

Because the heap is a binary tree, the height of any node is at most $\Theta(\lg n)$

Review: Heapify()

`Heapify()`: maintain the heap property

- Given: a node *i* in the heap with children *l* and *r*
- Given: two subtrees rooted at *l* and *r*, assumed to be heaps
- Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
  - ✓If A[i] < A[l] or A[i] < A[r], swap A[i] with the largest of A[l] and A[r]
  - ✓Recurse on that subtree
- Running time: O(*h*), h = height of heap = O(lg *n*)

Review: BuildHeap()

We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays

- Fact: for array of length *n*, all elements in range A[⌊n/2⌋ + 1 .. n] are heaps (*Why?*)
- So:
  - ✓Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
  - ✓Order of processing guarantees that the children of node *i* are heaps when *i* is processed

Review: BuildHeap()

```
// given an unsorted array A, make A a heap
BuildHeap(A)
{
  heap_size(A) = length(A);
  for (i = ⌊length[A]/2⌋ downto 1)
     Heapify(A, i);
}
```


Review: Priority Queues

Heapsort is a nice algorithm, but in practice Quicksort (coming up) usually wins

But the heap data structure is incredibly useful for implementing *priority queues*

- A data structure for maintaining a set *S* of elements, each with an associated value or *key*
- Supports the operations **`Insert()`**, **`Maximum()`**, and **`ExtractMax()`**
- ***What might a priority queue be useful for?***

Review: Priority Queue Operations

Insert(S, x) inserts the element x into set S

Maximum(S) returns the element of S with the maximum key

ExtractMax(S) removes and returns the element of S with the maximum key

Implementing Priority Queues

```
HeapInsert(A, key)    // what's running time?
{
    heap_size[A] ++;
    i = heap_size[A];
    while (i > 1  AND  A[Parent(i)] < key)
    {
        A[i] = A[Parent(i)];
        i = Parent(i);
    }
    A[i] = key;
}
```


Implementing Priority Queues

```
HeapExtractMax(A)
{
    if (heap_size[A] < 1) { error; }
    max = A[1];
    A[1] = A[heap_size[A]]
    heap_size[A] --;
    Heapify(A, 1);
    return max;
}
```

Review: Radix Sort

Radix sort:

- Assumption: input has *d* digits ranging from 0 to *k*
- Basic idea:
  - ✓Sort elements by digit starting with *least* significant
  - ✓Use a stable sort (like counting sort) for each stage
- Each pass over *n* numbers with *d* digits takes time O(*n*+*k*), so total time O(*dn*+*dk*)
  - ✓When *d* is constant and *k*=O(*n*), takes O(*n*) time
- **Fast! Stable! Simple!**
- Doesn't sort in place

Review: Bucket Sort

Bucket sort

- **Assumption: input is *n* reals from [0, 1)**
- **Basic idea:**
  - ✓**Create *n* linked lists (*buckets*) to divide interval [0,1) into subintervals of size 1/*n***
  - ✓**Add each input element to appropriate bucket and sort buckets with insertion sort**
- **Uniform input distribution → O(1) bucket size**
  - ✓**Therefore the expected total time is O(n)**
- **These ideas will return when we study *hash tables***

Address-Based Sorting

Proxmap uses techniques similar to hashing to assign an element to its correctly sorted position in a container such as an array using a hash function

The algorithms are generally complex, and very often only suitable for certain kinds of data

You need to find a suitable hashing function that will distribute data fairly evenly

Clashes are dealt with using a comparison based sort, so the more clashes there are the further the time complexity moves away from O(n) *(see notes view for more details)*

Radix / Bucket / Bin sort

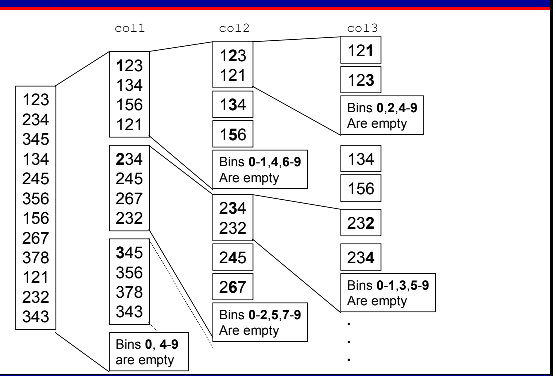
Radix Sort

- **In its favour, it is an *O(n)* sort. That makes it the fastest sort we have investigated.**
- **However it requires at least *2n* space in which to operate, and is in principle exponential in its space complexity..**
- **A radix sort makes one pass through the data for each *atom* in the key. If the key is three character uppercase alphabetic strings, such as ABC, then A is an atom. In this case, there would be three passes through the data.**

Cont..

- The first pass sorts on the low order element of the key (the **C** in our example). The second pass sorts on the next atom, in order of importance (the **B** in our example). Each pass progresses toward the high order atom of the key.
- In each such pass, the elements of the array are distributed into *k* buckets. For our alphabetic key example, **A**'s go into the 'A' bucket, **B**'s into the 'B' bucket, and so on. These distribution buckets are then gathered, in order, and placed back in the original array. The next pass is then executed. When a pass has been made on the high order atom in the key, the array will be sorted.

Radix Sort Example



What we studied

- ✓ Bubble Sort
- ✓ Selection Sort
- ✓ Insertion Sort
- ✓ Shell Sort
- ✓ Merge Sort
- ✓ Quick Sort
- ✓ Heap Sort
- ✓ Radix Sort

EXTERNAL SORTING

Learning Objectives

- External Sorting Techniques
    - K-way Merge Sort
    - Balanced Merge Sort
    - Poly-Phase Merge Sort

External Sorting

Need

- Entire data to be sorted might not fit in the available internal memory

Considerations

- When data resides in internal memory
  - ✓Data access time << Computation time
  - ✓Need to reduce the number of CPU operations
- When data resides on external storage devices
  - ✓Data access time >> Computation time
  - ✓Need to reduce disk accesses

Algorithms

Merge sort

Multi-way / k-way merge sort

- Balanced
- Poly-phase

General Approach

Divide data into smaller segments that can fit into internal memory

Sort them internally

Write the sorted segments (called *runs*) to secondary storage

Merge the *runs* together to get *runs* of larger size

Continue until a single *run* is left

External- Merge sort

Also called 2-way merge sort

Assumptions
- There are N records on the disk
- It is possible to sort M records using internal sort (at a time)

External- Merge Sort

Sort process
- Create N/M sorted runs, reading M records at a time
- Set aside 3 blocks of internal memory each capable of holding M/3 records
- First two blocks act as input buffers
- Third acts as output buffer
- Merge runs {R1, R2}; {R3, R4} to get N/2M runs of size 2M each
- Continue merging till a single run of size N is not obtained

Merging of blocks

To do: merge two blocks B1, B2 using a block B3
Set I<- 1, J<-1
If Key_B1_I < Key_B2_J
    Write Rec_B1_I to B3
    Increment I
Else
    Write Rec_B2J to B3
    Increment J
If B3 is full flush it on disk
If B1 is empty read next block from R1
If B2 is empty read next block from R2
Result a run R3; Size(R3) = Size (R1) + Size (R2)

Multi-way Merge / K-way merge

Number of passes for a 2 way merge: $\log_2(N/M)$

We can reduce the number of passes by using a higher order merge

Thus, if a merge of order K is used then number of passes $\log_k(N/M)$

Consideration

As the number of comparisons to be made increases there is a small overhead in terms of CPU computation

Generally a heap of leading values from each run/block is maintained

Merging with Tapes

Limitations

- Only sequential access possible
- Reading from multiple runs simultaneously would require multiple tape drives
- Lesser number of drives would decrease the time efficiency

2-Way Merge (with Tape Drives)

Assumptions

- Available number of tape drives: 4 (2*2)
- Say the tapes are named U, V, W, X
- All the data is initially on tape U
- Internal memory can sort M records at a time
- Total number of records is N

Depending upon the pass number the pair (U,V) or (W,X) can act either as a set of input tapes or output tapes

Cont..

Read M records from U
Sort them internally and Write them alternately to W/X
Do
Merge Ith run from W with Ith run on X; Write to U
Merge (I+1)th run from W with (I+1)th run on X; Write to V
Continue till all runs are not processed

Result:

N/2M runs of length 2M each, placed alternately on tapes W & X
W & X become the input tapes
U & V become the output tapes

Repeat the merge process till you don't get a single run of length N

Cont..

Set I<- 1
Start
Merge Ith runs from Input Tape 1 & Input Tape 2
Place the result on Output Tape 1
Set I<- I+1
Merge Ith runs from Input Tape 1 & Input Tape 2
Place the result on Output Tape 2
Set I<- I+1
Continue till all the runs are not processed

Result:

N/2M runs of length 2M each
Repeat the process after inverting the role of tapes till you don't get a single run of N records

Cont..

Say total number of passes is P
M * (2 * 2 * 2 … P times) = N
$M * 2^P = N$
$\text{Log}_2 (N/M) = P$

Illustration

Given:

- Four tapes one of which contains the data to be sorted
- Number of records that can be held and sorted in main memory at a time: 3

T1	2	12	3	5	23	6	50	7	31	90	22	11	15	78	45	40
T2																
T3																
T4																

After Pass I

T1																
T2																
T3	2	3	12	7	31	50	15	45	78							
T4	5	6	23	11	22	90	40									

After Pass II

T1	2	3	5	6	12	23	15	40	45	78						
T2	7	11	22	31	50	9										
T3																
T4																

After Pass III

T1																
T2																
T3	2	3	5	6	7	11	12	22	23	31	50	90				
T4	15	40	45	78												

After Pass IV

T1	2	3	5	6	7	11	12	15	22	23	31	40	45	50	78	90
T2																
T3																
T4																

K-way Merge Sort

Motivation

Need for better performance

Strategy

Increase the number of runs that are merged at a time

Say, K runs are merged at a time

Requirement

According to the above algorithm we require K input tapes and K output tapes => 2*K number of tape drives

Number of passes

$\text{Log}_k (N/M) = P$

Illustration: 3-Way Merge

Requirement 2*3 Tapes Say, M=3

T1	2	12	3	5	23	6	50	7	31	90	22	11	15	78	45	40
T2																
T3																
T4																
T5																
T6																

After Pass I

Runs of size M (3) distributed on tapes T4, T5 and T6

T1																
T2																
T3																
T4	2	3	12	11	22	90										
T5	5	6	23	15	45	78										
T6	7	31	50	40												

After Pass II

Action: Corresponding runs from the three input tapes T4, T5 and T6 merged and placed on T1, T2 and T3 respectively.
Result: Runs of size 3*3 = 9

T1	2	3	5	6	7	12	23	31	50							
T2	11	15	22	40	45	90	78									
T3																
T4																
T5																
T6																

After Pass III

Action: Corresponding runs from the three input tapes T1, T2 and T3 merged and placed on T4, T5 and T6 respectively.
Result: Runs of size 3*6 = 18

T1																
T2																
T3																
T4	2	3	5	6	7	11	12	15	22	23	31	40	45	50	78	90
T5																
T6																

K-Way Merge Sort

K-Way Sorting can also be implemented using K+1 tapes

Mechanism:

Use K tapes as input tapes and one as output tape.

After i^{th} pass, place the runs of length 2^i*M on the output tape

Redistribute the runs on the input tapes

Drawback:

An additional pass over the output tape to redistribute the runs onto K-tapes for the next level

Balanced Merge Sorts

The sorting technique used so far is ***Balanced Merge Sort***

Characteristic

An even distribution of runs onto K-input tapes

Result

Either 2K tapes are required

Or extra passes for redistribution of data are required

Solution

Use uneven distribution

Poly-Phase Merge Sort

Technique

Uses uneven distribution of runs over K input tapes

Requirement:

K Input tapes + 1 Output tape

Basis of distribution:

Fibonacci numbers

Poly-Phase Merge Sort

Say 21 runs are to be merged using 3 tapes.
Contents of tapes after each phase:

	Init	After T2+T3	After T1+T2	After T1+T3	After T2+T3	After T1+T2	After T1+T3
T1	0	8	3	0	2	1	0
T2	13	5	0	3	1	0	1
T3	8	0	5	2	0	1	0

What we Studied

- ✓ External Sorting Techniques
  - ✓K-way Merge Sort
  - ✓ Balanced Merge Sort
  - ✓ Poly-Phase Merge Sort

Conclusion

- ✓ Searching Techniques
- ✓ Internal Sorting Techniques
- ✓ External Sorting Techniques

Review Questions (Objective)

1. A list is ordered from smaller to largest when a sort is called. Which sort would take the longest time to execute?
2. A list is ordered from smaller to largest when a sort is called. Which sort would take the shortest time to execute?
3. When will you sort an array of pointers to list elements, rather than sorting the elements themselves?
4. The element being searched for is not found in an array of 100 elements. What is the average number of comparisons needed in a sequential search to determine that the element is not there, if the elements are completely unordered?
5. What is the average number of comparisons needed in a sequential search to determine the position of an element in an array of 100 elements, if the elements are ordered from largest to smallest?
6. Which sort show the best average behavior?
7. What is the average number of comparisons in a sequential search?
8. Which one is faster? A binary search of an orderd set of elements in an array or a sequential search of the elements.
9. Under what circumstances would you not use a quick sort.
10. Define Heap sort with example.
11. Running time of merge sort algrothim is...........
12. Define radix sort.

Review Questions (Short Type)

1. Compare the time complexity of various sorting algrothims (Best and Worst case).
2. Write an algrothim (non-recursive) to implement quick sort.
3. What do u mean by insertion sort.Which data structure is best suited for inserted sort and why.Explain the algorithm for insertion sort.
4. What do u mean by searching.What are the condition for binary search.Explain the algorithm for the binary search.
5. Write a Binary Search program
6. Give the difference between linear and binary search.
7. Define merge sort .give its algorithm
8. Write programs for Bubble Sort, Quick sort
9. List out few of the Application of tree data structure
10. Define selection sort with example.
11. What is heap sort? Explain with example.
12. Explain binary search and linear search.
13. Explain merge sort with example.
14. Give the complexity of all sorting algorithms.
15. Give the complexity of binary search algorithm .explain it
16. Give the limitations of binary search algorithm.
17. Under what circumstances would you not use a quick sort.
18. Sort the given values using Quick Sort?
19. 657075808560555045

Review Questions (Long Type)

1. Explain quick sort and merge sort algorithms and derive the time-constraint relation for these.
2. Write algorithm for any of the following sorting methods: -
   - Merge Sort
   - Quick Sort
   - Radix Sort
3. Compare above three methods of sorting for ideal, worst and average cases.
4. Define bubble sort. Give the algorithm and explain it with example.
   Suppose the following numbers are stored in an array A.
   32, 51, 27, 85 , 66, 23, 13, 57.
   Sort the array using bubble sort.
5. What do u mean by hashing.What are various hash fuction.Explain three hash function.
6. What do u mean by searching.What are various searching techniques.Which searching technique u like the most and why.Give an algorithm for the searching technique.
7. Write an algorithm for radix sort.
8. Define linked list.Give the algorithms when list is sorted and unsorted.

References

- **"Fundamentals of Data Structures", E. Horowitz and S. Sahani, Galgotia Booksource Pvt. Ltd., (1999)**
- ***Data Structures and Algorithm Analysis in C (Second Edition) by Mark Allen Weiss***
- ***Data Structures: A Pseudocode Approach with C, Second Edition Richard Gilberg, Behrouz Forouzan***
- **"Data Structures and program design in C", R. L. Kruse, B. P. Leung, C. L. Tondo, PHI.**
- **"Data Structure", Schaum's outline series, TMH, 2002**
- **"Data Structures using C and C++", Y. Langsam et. al., PHI (1999).**
- **"Data Structures", N. Dale and S.C. Lilly, D.C. Heath and Co. (1995).**
- **"Data Structure & Algorithms", R. S. Salaria, Khanna Book Publishing Co. (P) Ltd., 2002.**