Merge Sort

The merge sort algorithm is defined recursively:

- If the list is of size 1, it is sorted—we are done;
- Otherwise:
  - Divide an unsorted list into two sub-lists,
  - Sort each sub-list recursively using merge sort, and
  - Merge the two sorted sub-lists into a single sorted list
- Divide and conquer algorithm

Merge Sort (2)

Need to sort:

34 8 64 51 32 21

Sort 34 8 64 : 8, 34, 64

Sort 51 32 21: 21, 32, 51

Merge the two:

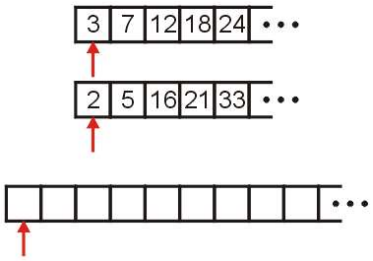
We have 8, 21, 32, 34,51,64

Question: How quickly can we recombine the two sub-lists into a single sorted list?

Merging Example

Consider the two sorted arrays and an empty array

Define three indices at the start of each array

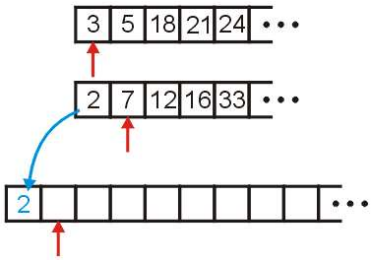


Merging Example

We compare 2 and 3: 2 < 3

– Copy 2 down

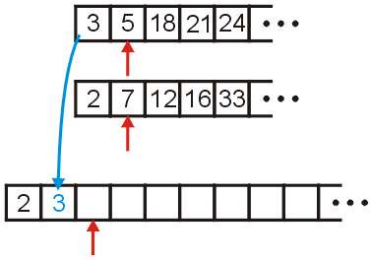
– Increment the corresponding indices



Merging Example

We compare 3 and 7

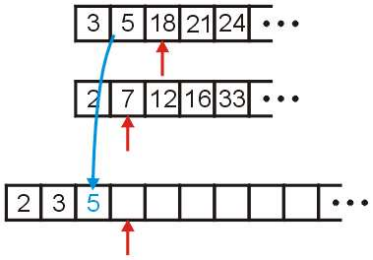
- Copy 3 down
- Increment the corresponding indices



Merging Example

We compare 5 and 7

- Copy 5 down
- Increment the appropriate indices

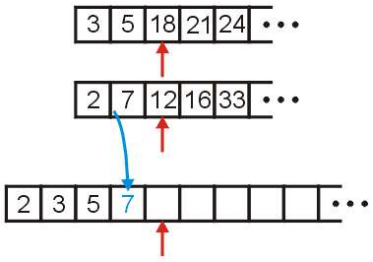


Merging Example

We compare 18 and 7

– Copy 7 down

– Increment...

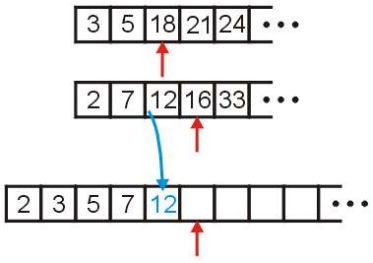


Merging Example

We compare 18 and 12

– Copy 12 down

– Increment...

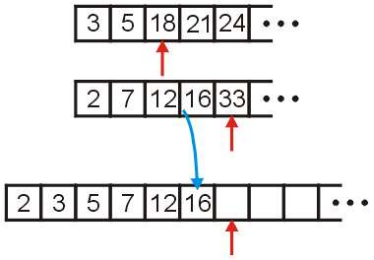


Merging Example

We compare 18 and 16

– Copy 16 down

– Increment...

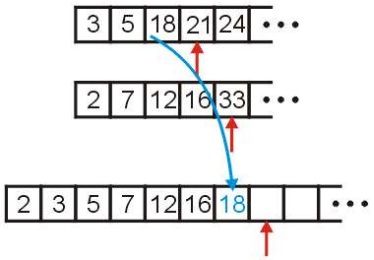


Merging Example

We compare 18 and 33

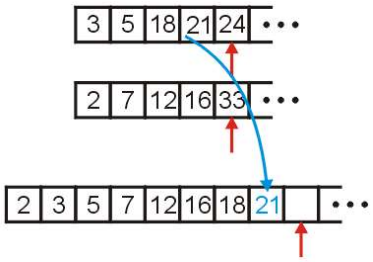
– Copy 18 down

– Increment...



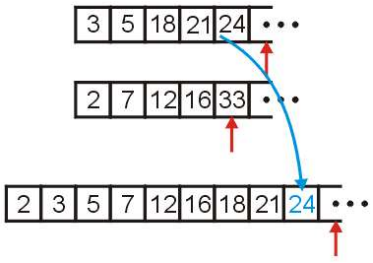
Merging Example

We compare 21 and 33
- Copy 21 down
- Increment...



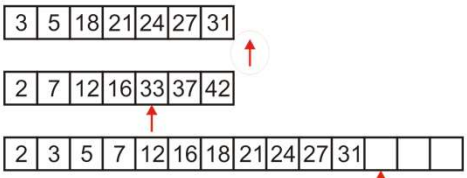
Merging Example

We compare 24 and 33
- Copy 24 down
- Increment...

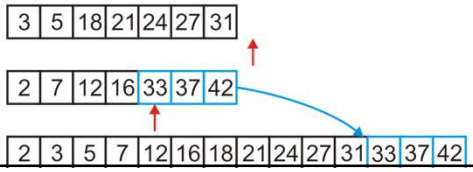


Merging Example

We would continue until we have passed beyond the limit of one of the two arrays



After this, we simply copy over all remaining entries in the non-empty array



Merging Two Lists

Programming a merge is straight-forward:

- the sorted arrays, `array1` and `array2`, are of size `n1` and `n2`, respectively, and
- we have an empty array, `arrayout`, of size `n1 + n2`

Define three variables

```
int i1 = 0, i2 = 0, k = 0;
```

which index into these three arrays

Merging Two Lists

We can then run the following loop:

```
//...
int i1 = 0, i2 = 0, k = 0;

while ( i1 < n1 && i2 < n2 ) {
    if ( array1[i1] < array2[i2] ) {
        arrayout[k] = array1[i1];
        ++i1;
    } else {

        arrayout[k] = array2[i2];
        ++i2;
    }
    ++k;
}
```

Merging Two Lists

We're not finished yet, we have to empty out the remaining array

```
for ( ; i1 < n1; ++i1, ++k ) {
    arrayout[k] = array1[i1];
}

for ( ; i2 < n2; ++i2, ++k ) {
    arrayout[k] = array2[i2];
}
```

Analysis of merging

- The body of the loops run a total of $n_1 + n_2$ times
- Hence, merging may be performed in $\Theta(n_1 + n_2) = \Theta(n)$ time

Problem: We cannot merge two arrays in-place

- This algorithm always required the allocation of a new array
- Therefore, the memory requirements are also $\Theta(n)$

The Merge Sort Algorithm

The algorithm:

- Split the list into two approximately equal sub-lists
- Recursively call merge sort on both sub lists
- Merge the resulting sorted lists

The Merge Sort Algorithm

Question:

- we split the list into two sub-lists and sorted them
- how should we sort those lists?

Answer (theoretical):

- if the size of these sub-lists is > 1, use merge sort again
- if the sub-lists are of length 1, do nothing: a list of length one is sorted

Code of Merge Sort

```
void
MSort( ElementType A[ ], ElementType TmpArray[ ],
                int Left, int Right )
{
    int Center;

    if( Left < Right )
    {
        Center = ( Left + Right ) / 2;
        MSort( A, TmpArray, Left, Center );
        MSort( A, TmpArray, Center + 1, Right );
        Merge( A, TmpArray, Left, Center + 1, Right );
    }
}

void
Mergesort( ElementType A[ ], int N )
{
    ElementType *TmpArray;

    TmpArray = malloc( N * sizeof( ElementType ) );
    if( TmpArray != NULL )
    {
        MSort( A, TmpArray, 0, N - 1 );
        free( TmpArray );
    }
    else
        FatalError( "No space for tmp array!!!" );
}
```

Figure 7.9 Mergesort routine

```
/* Lpos = start of left half, Rpos = start of right half */

void
Merge( ElementType A[ ], ElementType TmpArray[ ],
                int Lpos, int Rpos, int RightEnd )
{
    int i, LeftEnd, NumElements, TmpPos;

    LeftEnd = Rpos - 1;
    TmpPos = Lpos;
    NumElements = RightEnd - Lpos + 1;

    /* main loop */
    while( Lpos <= LeftEnd && Rpos <= RightEnd )
        if( A[ Lpos ] <= A[ Rpos ] )
            TmpArray[ TmpPos++ ] = A[ Lpos++ ];
        else
            TmpArray[ TmpPos++ ] = A[ Rpos++ ];

    while( Lpos <= LeftEnd )  /* Copy rest of first half */
        TmpArray[ TmpPos++ ] = A[ Lpos++ ];
    while( Rpos <= RightEnd ) /* Copy rest of second half */
        TmpArray[ TmpPos++ ] = A[ Rpos++ ];

    /* Copy TmpArray back */
    for( i = 0; i < NumElements; i++, RightEnd-- )
        A[ RightEnd ] = TmpArray[ RightEnd ];
}
```

Figure 7.10 *Merge* routine

Run-time

The following table summarizes the run-times of merge sort

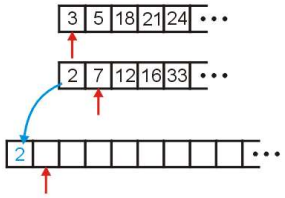
T(n) = 2T(N/2) + n => T(n) = nlogn

Case	Run Time	Comments
Worst	$\Theta(n \log(n))$	No worst case
Average	$\Theta(n \log(n))$	
Best	$\Theta(n \log(n))$	No best case

Why is it not O(n^2)

When we are merging, we are comparing values

- What operation prevents us from performing O(n^2) comparisons?
- During the merging process, if 2 came from the second half, it was only compared to 3 and it was not compared to any other of the other n – 1 entries in the first array



Space Complexity

- Addition Memory required is O(logn +n) = O(n)
  - Each recursive function call places its local variables, parameters, *etc.*, on a stack
    - The depth of the recursion tree is O(log*n*)
  - Temporary array for merging: O(n)

- It is hardly ever used for main memory sorts
  - Space complexity
  - Additional work spent copying temporary array.

- We move on Quicksort, a recursive algorithm which can be done ***almost*** in-place

Quiz

- Sort 3, 1,4, 1,5,9,2,6 using mergesort.

Quicksort: idea

Merge sort splits the array sub-lists and sorts them

The larger problem is split into two sub-problems based on ***location*** in the array

Consider the following alternative:

- Chose an object in the array and partition the remaining objects into two groups relative to the chosen object => **Quicksort**
- **Chosen object is called pivot**

Quick Sort: idea (2)

1. We will choose a pivot element in the list.
2. Partition the array into two parts.
3. One contains elements smaller than this pivot, another contains elements larger than this pivot.
4. Recursively sort and merge the two partitions.

How do we merge?

We do not need any additional space for merging. Thus additional storage is O(1)

Quicksort

For example, given

80	38	95	84	66	10	79	44	26	87	96	12	43	81	3

we can select the middle entry, 44, and sort the remaining entries into two groups, those less than 44 and those greater than 44:

38	10	26	12	43	3	44	80	95	84	66	79	87	96	81

Notice that 44 is now in the correct location if the list was sorted

– Proceed by applying the algorithm to the first six and last eight entries

Worst-case scenario

Moral of the story: We would like the partitions to be as equal as possible.

This depends on the choice of the pivot => Quicksort running time depends on the pivot

Equal partitions are assured if we can choose our pivot as the element which is in the middle of the sorted list.

It is not easy to find that.

Pivots are chosen randomly.

Worst case complexity?

Worst-case scenario

Suppose we choose the first element as our pivot and we try ordering a sorted list:

80	38	95	84	66	10	79	2	26	87	96	12	43	81	3

Using 2, we partition into

2	80	38	95	84	66	10	79	26	87	96	12	43	81	3

We still have to sort a list of size $n - 1$

The run time is $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

– Thus, the run time drops from $n \ln(n)$ to n^2

Worst-case scenario

Our goal is to choose the median element in the list as our pivot:

80	38	95	84	66	10	79	2	26	87	96	12	43	81	3

Unfortunately, it's difficult to find. Pivot is usually chosen randomly

Alternate strategy: take the median of a subset of entries

- For example, take the median of the first, middle, and last entries

Median-of-three

It is difficult to find the median so consider another strategy:

- Choose the median of the first, middle, and last entries in the list

This will usually give a better approximation of the actual median

Partitioning process

Interchange the pivot with the last element

Have a pointer at the first element (P1), and one at the second last element (P2).

Move P1 to the right skipping elements which are less than the pivot.

Move P2 to the left skipping elements which are more than the pivot.

Stop P1 when we encounter an element greater than or equal to the pivot.

Stop P2 when we encounter an element lesser than or equal to the pivot.

Partitioning process (2)

Interchange the elements pointed to by P1 and P2.

If P1 is right of P2, stop, otherwise move P1 and P2 as before till we stop again.

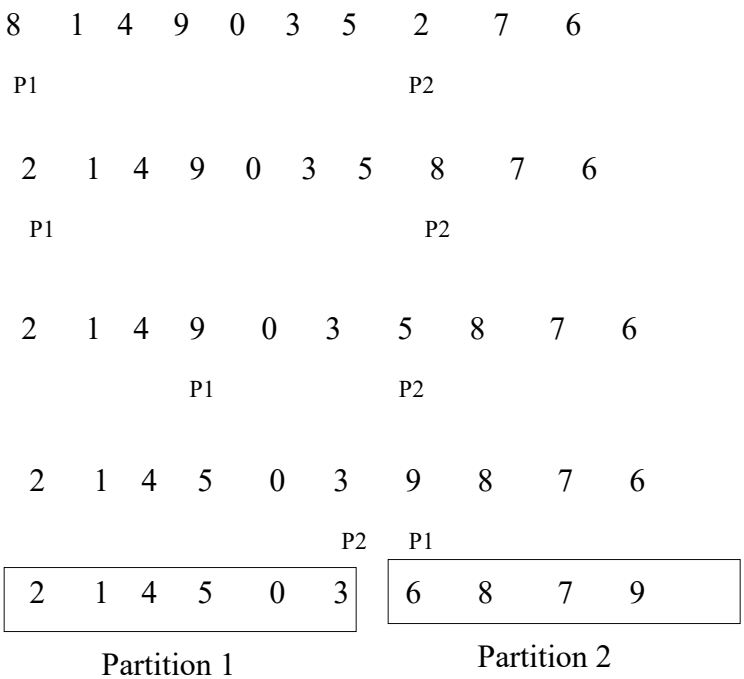
When we stop, swap P1 with the last element which is the pivot

8 1 4 9 6 3 5 2 7 0

First = 8 Last = 0, Median = 6, Pivot = 6

8 1 4 9 0 3 5 2 7 6

P1 P2



At any time can you say anything about the elements to the left of P1?

Elements to the left of P1 are less than or equal to the pivot.

Also, for right of P2?

Elements to the right of P1 are greater than or equal to the pivot.

When P1 and P2 cross, what can you say about the elements in between P1 and P2?

They are all equal to the pivot.

Suppose P1 and P2 have crossed, and stopped and the pivot is interchanged with P1.

How do we form the partition?

Everything including P1 and its right are in one partition (greater).

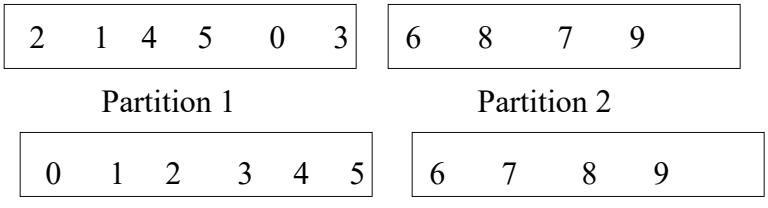
Remaining are in the left partition.

Procedure Summary

Partition the array

Sort the partition recursively.

8 1 4 9 6 3 5 2 7 0



Need to do any thing more?

Merger is automatic

Pseudocode

```
Quicksort(A, left, right)
   { Find pivot;
      Interchange pivot and A[right];
      P1 = left; P2 = right – 1;
      Partition(A, P1, P2, pivot);   /*returns newP1*/
       Interchange A[newP1] and A[right];
      Quicksort(A, left, newP1-1);
       Quicksort(A,  newP1, right);  }
```

```
Partition(A, P1, P2,pivot)
   {
       While  (P1 ≤ P2)
          {
              While (A[P1] < pivot)   increment P1;
               While (A[P2] > pivot)   decrement P2;
                 Swap A[P1] and A[P2];
                 increment P1;  decrement P2;
          }
      newP1 = P1;
      return(newP1);
}
```

Run time

$T(n) = T(n_1) + T(n_2) + cn$

$T(1) = 1;$

$n_1 + n_2 = n$

In good case, $n_1 = n_2 = n/2$ always

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

Run time (2)

In worst case, $n_1 = 1$ $n_2 = n-1$ always

$T(n) = pn + T(n-1)$

$= pn + p(n-1) + T(n-2)$

……..

$= p(n + n-1 + \ldots\ldots + 1)$

$= pn^2$

Thus $T(n)$ is $O(n^2)$ in worst case.

Average case complexity is $O(n\log n)$

Memory Requirements

The additional memory required is $\Theta(\ln(n))$

– Each recursive function call places its local variables, parameters, *etc.*, on a stack
  - The depth of the recursion tree is $\Theta(\ln(n))$

– Unfortunately, if the run time is $\Theta(n^2)$, the memory use is $\Theta(n)$

Run-time Summary

To summarize all three algorithms

	Best Run Time	**Average Run Time**	**Worst-case Run Time**	**Average Memory**	**Worst-case Memory**
Merge Sort		$\Theta(n\log(n))$		$\Theta(n)$	
Quicksort	$\Theta(n\log(n))$	$\Theta(n\log(n))$	$\Theta(n^2)$	$\Theta(\log(n))$	$\Theta(n)$
Insertion Sort	$\Theta(n)$	$\Theta(n+\mathrm{d})$	$\Theta(n^2)$	$\Theta(1)$	

Comments

Quicksort performs well for large inputs, but not so good for small inputs.

When the divisions become small, we can use insertion sort to sort the small divisions instead.

5 <= cutoff <= 20

Is This The Best We Can Do?

- Sorting by Comparison
  - Only information available to us is the *set of N items* to be sorted
  - Only operation available to us is *pairwise comparison between 2 items*

General Lower Bound For Sorting is **Ω(nlog n)**

Is This The Best We Can Do?

- Sorting by Comparison
  - Only information available to us is the *set of N items* to be sorted
  - Only operation available to us is *pairwise comparison between 2 items*

What happens if we relax these constraints?

Special Case Sorting

Now we will present a linear time sorting algorithm: Bin sort or Bucket sort.

These apply only when the input has special constraints, e.g., inputs are integers and we already know the min and max of each element.

BinSort (a.k.a. BucketSort)

Requires:

- Having an array with N elements
- Each element is in {1, …, K}

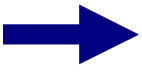
Works by:

Putting items into correct bin (cell) of array, based on key

BinSort example

K=5 list=(5,1,3,4,3,2,1,1,5,4,5)

Bins in array	
key = 1	1,1,1
key = 2	2
key = 3	3,3
key = 4	4,4
key = 5	5,5,5

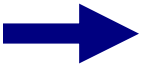


Sorted list:
1,1,1,2,3,3,4,4,5,5,5

BinSort example (2)

- K=5 list=(5,1,3,4,3,2,1,1,5,4,5)

Bins in array	
key = 1	3
key = 2	1
key = 3	2
key = 4	2
key = 5	3



Sorted list:
1,1,1,2,3,3,4,4,5,5,5

BinSort Pseudocode

```
procedure BinSort (List L,K)

 bins[1..K][]
// Each element of array bins is linked list.
// Could also BinSort with array of arrays.

For Each number x in L
      bins[x].Append(x)
End For
For i = 1..K
      For Each number x in bins[i]
            Print x
      End For
End For
```

BinSort Running Time

- K is a constant
  - BinSort is linear time
  - O(n + K) = O(n)
- K is variable
  - Not simply linear time
- K is very large (e.g. 2^{32})
  - Impractical
- Storage: O(K) or O(n)
  - Storage could be large for large K.

Summary

- Merge Sort
- Quick Sort
- Bin Sort (Bucket Sort)

- Next week: Online Assignment 1& Searching
  - **Content**: Algorithm analysis and Sort Algorithms
  - **Duration**: 1h
  - You will do your exam by using your laptop. After finishing, you will submit your answers to specific folder on ILIAS (elearning.vgu.edu.vn)