Sorting

- a list of records (R_1, R_2, …, R_n)
- each R_i has key value K_i
- assume an ordering relation (<) on the keys, so that for any 2 key values x and y, x=y or x<y or x>y. < is transitive.

The sorting problem is that of finding a permutation, σ, such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}$, $1 \leq i \leq n-1$. The desired ordering is ($R_{\sigma(1)}$, $R_{\sigma(2)}$, …, $R_{\sigma(n)}$).

Stable Sorting

Let σ_s be the permutation with the following properties:

(1) $K_{\sigma s(i)} \leq K_{\sigma s(i+1)}$, $1 \leq i \leq n-1$.

(2) If $i<j$ and $K_i = K_j$ in the input list, then R_i precedes R_j in the sorted list.

A sorting method that generates the permutation σ_s is **stable**.

Two main operations

- Key Compare
- Data movement

Sorting Categories

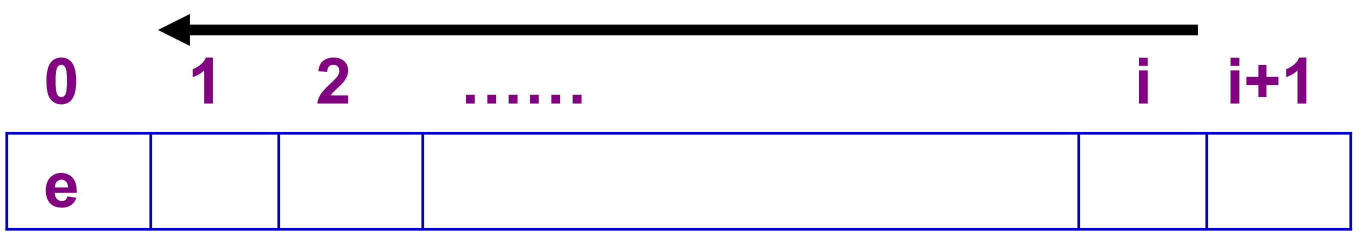
- Data location
  - Internal Sorting
  - External Sorting
- Sorting principle
  - Insert
  - Exchange
  - Selection
  - Merge
  - Multiple keys

- Sorting
  - internal
    - insert
    - quick
    - merge
    - heap
    - radix
  - external
    - m-way merge
    - run generation

Assume that relational operators have been overloaded so that record comparison is done by comparing their keys

Insert Sort

- Basic step : Insert **e** into a sorted sequence of **i** records in such a way that the resulting sequence of size **i+1** is also ordered



- Uses a[0] to simplify the while loop test

```
template<class T> viod Insert(const T& e, T *a, int i)
{// a must have space for at least i+2 elements.
    a[0]=e;
    while (e < a[i])
    {
        a[i+1]=a[i];
        i--;    // a[i+1] is always ready for storing element
    }
    a[i+1]=e;
}
```

- **Insertion sort:**
- Begin with the ordered sequence a[1], then successively insert a[2], a[3], …, a[n] into the sequence.
- **template**<**class** T>**void** InsertionSort (Element *list, **const int** n)
- { //sort a[1:n] into nondescreasing order.
-     **for (int** j=2**;** j<=n**;** j++) {
-         T temp = a[j]**;**
-         Insert (temp, a, j-1)**;**
-     }
- }

Example

i=1 (49) 38 65 97 76 13 27

i=2 38 (38 49) 65 97 76 13 27

i=3 65 (38 49 65) 97 76 13 27

i=4 97 (38 49 65 97) 76 13 27

i=5 76 (38 49 65 76 97) 13 27

i=6 13 (13 38 49 65 76 97) 27

i=7 27 (13 27 38 49 65 76 97

j j j j j j

Result: (13 27 38 49 65 76 97)

Analysis of insert sort

The worst case

Insert(e, a, i) makes i+1 comparisons before making insertion --- O(i).

InsertSort invokes Insert for i=j-1=1, 2,…,n-1, so the overall time is

$$O\left(\sum_{i=1}^{n-1}(i+1)\right)=O(n^2).$$

Insert sort

Variations:

Linked Insert Sort

Binary Insert Sort

Shell Insert Sort

Binary Insert Sort

i=1 (30) 13 70 85 39 42 6 20

i=2 13 (13 30) 70 85 39 42 6 20

⋮

i=7 6 (6 13 30 39 42 70 85) 20

i=8 20 (6 13 30 39 42 70 85) 20

s m j

i=8 20 (6 13 30 39 42 70 85) 20

s m j

i=8 20 (6 13 30 39 42 70 85) 20

s mj

i=8 20 (6 13 30 39 42 70 85) 20

j s

i=8 20 (6 13 20 30 39 42 70 85)

```
void binsort(JD r[],int n) {
  int i,j,x,s,m,k;
  for(i=2;i<=n;i++) {
    r[0]=r[i];
    x=r[i].key;
    s=1; j=i-1;
    while(s<=j) {
      m=(s+j)/2;
      if(x<r[m].key)
           j=m-1;
      else
           s=m+1;
    }
    for(k=i-1;k>=s;k--)
      r[k+1]=r[k];
    r[s]=r[0];
  }
}
```

Shell Insert Sort

- Donald Shell
- Diminishing increment sort
  - Select an integer d1=h < n,
    - every *h*th elements yields a group
    - Sort the groups via simple insert sort
    - Results to h-sorted file
  - Select an integer d2< h
    - Grouping &sorting
  - Until di = 1

49 38 65 97 76 13 27 48 55 4

d1=5

Grouping: 49 38 65 97 76 13 27 48 55 4

Sorting: 13 27 48 55 4 49 38 65 97 76

d2=3

Grouping: 13 27 48 55 4 49 38 65 97 76

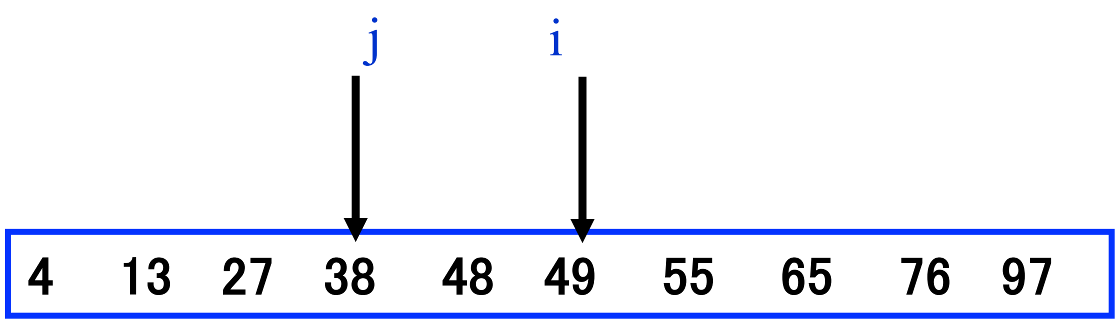
Sorting: 13 4 48 38 27 49 55 65 97 76

d3=1

Grouping: 13 27 48 55 4 49 38 65 97 76

Sorting: 4 13 27 38 48 49 55 65 76 97

```
void shellsort(JD r[],int n,int d[T]) {
   int i,j,k;
   JD x; k=0;
   while(k<T) {
      for(i=d[k]+1;i<=n;i++) {
         x=r[i];
         j=i-d[k];
         while((j>0)&&(x.key<r[j].key)) {
            r[j+d[k]]=r[j];
            j=j-d[k];
         }
         r[j+d[k]]=x;
      }
      k++;
   }
}
```



```
#define T 3
int d[]={5,3,1};
```

Round 1   13 27 48 55 4 49 38 65 97 76

j j j j j i i i i i

Round 2 : 13 4 48 38 27 49 55 65 97 76

j j j j j i i j i j i j i j i i i

Round3 : 13 4 48 38 27 49 55 65 97 76

Shell Insert Sort

- Selection of increments (di)
- ???

- **Exercises: P401-1, 3**

Exchange Sorting: Bubble Sorting

	Round one	Round two	Round three	Round four	Round five	Round six
38	38	38	38	13	13	13
49	49	49	13	27	27	27
65	65	13	27	30	30	30
76	13	27	30	38	38	
13	27	30	49	49		
27	30	65	65			
30	76	76				
97	97					

```
void bubble_sort(JD r[],int n) {
  int m,i,j,flag=1;
  JD x;
  m=n-1;
  while((m>0)&&(flag==1)) {
    flag=0;
    for(j=1;j<=m;j++)
      if(r[j].key>r[j+1].key) {
        flag=1;
        x=r[j];
        r[j]=r[j+1];
        r[j+1]=x;
      }
    m--;
  }
}
```

- Cocktail sort

```
function cocktail_sort(list, list_length){
 bottom = 0; top = list_length - 1;
 swapped = true;
 while(swapped == true) {
    swapped = false;
    for(i = bottom; i < top; i = i + 1) {
       if(list[i] > list[i + 1]) {
           swap(list[i], list[i + 1]);
           swapped = true;
       }
    }
   top = top - 1;
   for(i = top; i > bottom; i = i - 1) {
      if(list[i] < list[i - 1]) {
          swap(list[i], list[i - 1]);
          swapped = true;
      }
   }
 bottom = bottom + 1;
 }
}
```

Quick Analysis

- M runs/rounds
- Each run : O(n)
  - One record selected
  - Traverse the whole remaining unsorted file
- How to improve?
  - Each run: traverse part of the file
  - How?

[a1 a2 a3 a4 a5 a6 a7 a8] →

Mid = a1 →

[a2 a3 a4] a1 [a5 a6 a7 a8]

Quick Sort

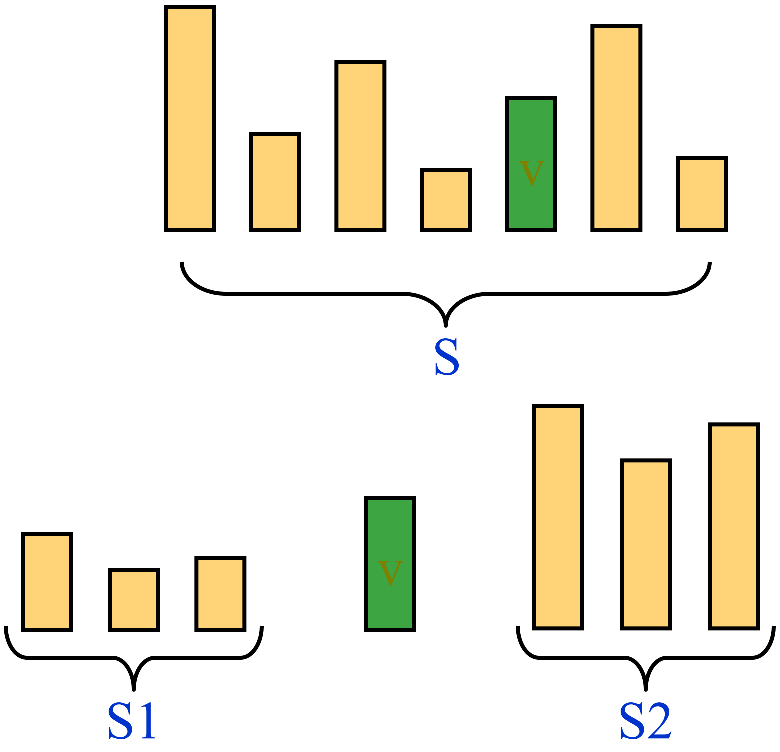
- Fastest known sorting algorithm in practice
- Average case: O(n log n)
- Worst case: $O(n^2)$
  - But, the worst case seldom happens.
- Divide-and-conquer recursive algorithm

Quicksort

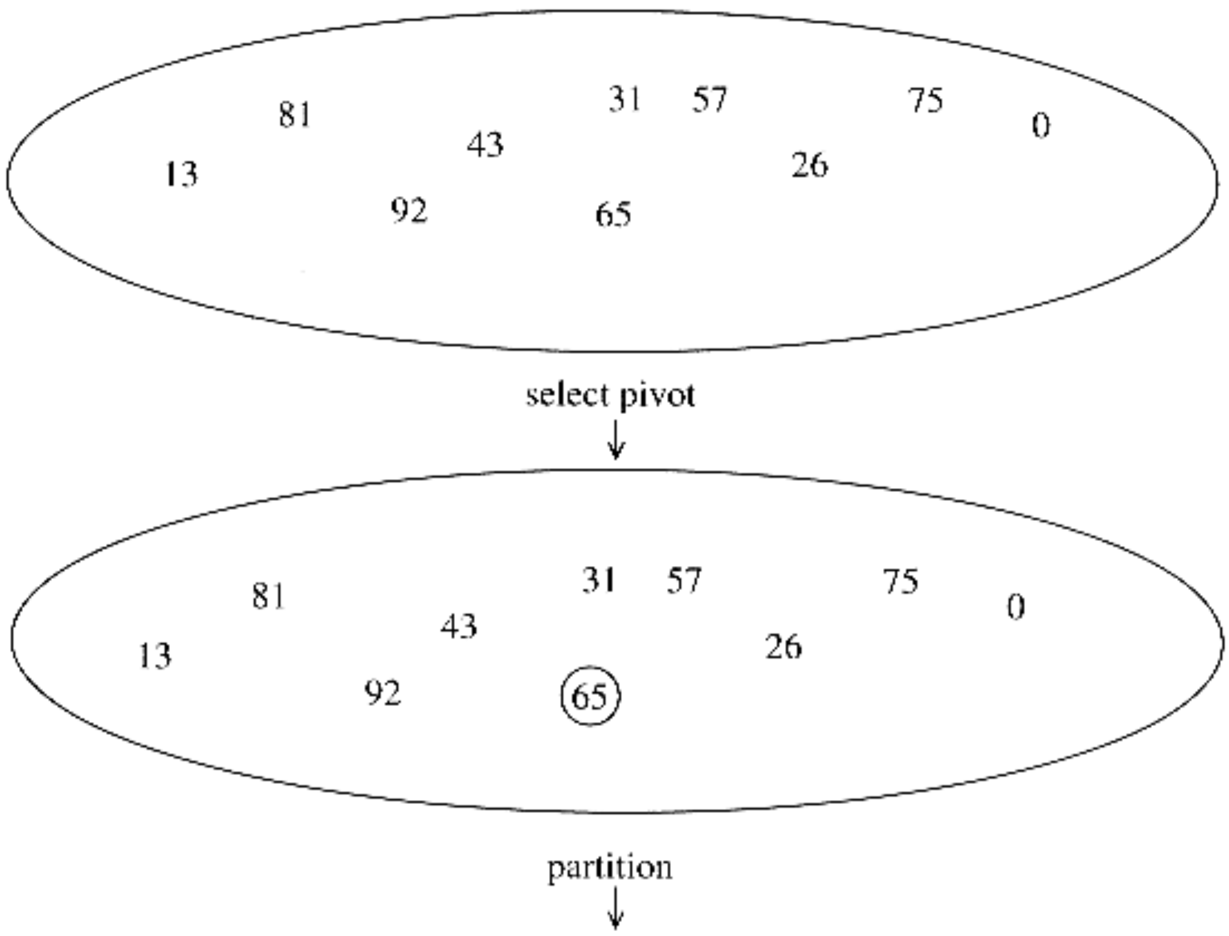
- **Divide step:**
  - **Pick any element (*pivot*) v in S**
  - **Partition S – {v} into two disjoint groups**

    $S1 = \{x \in S - \{v\} \mid x \le v\}$

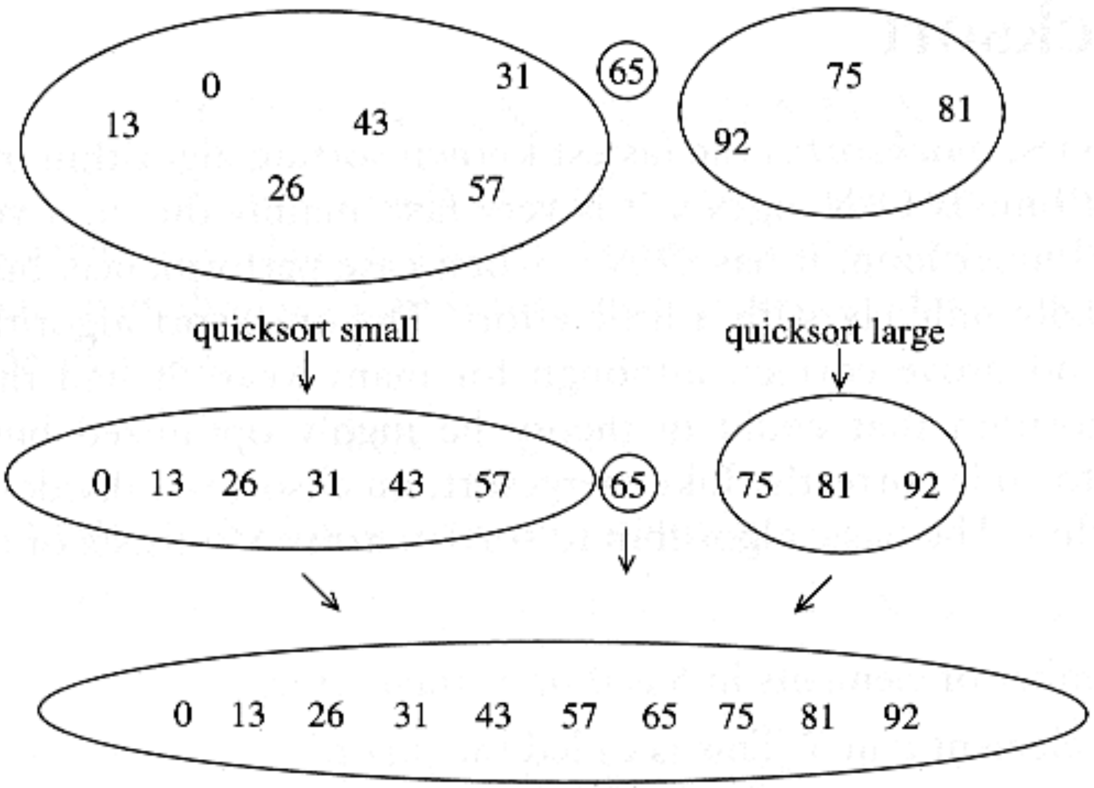
    $S2 = \{x \in S - \{v\} \mid x \ge v\}$
- **Conquer step**
  - **Recursively sort S1 and S2**
- **Combine step**
  - **Combine the sorted S1, followed by v, followed by the sorted S2**



Example: Quicksort



Example: Quicksort...



Pseudocode

Input: an array A[p, r]

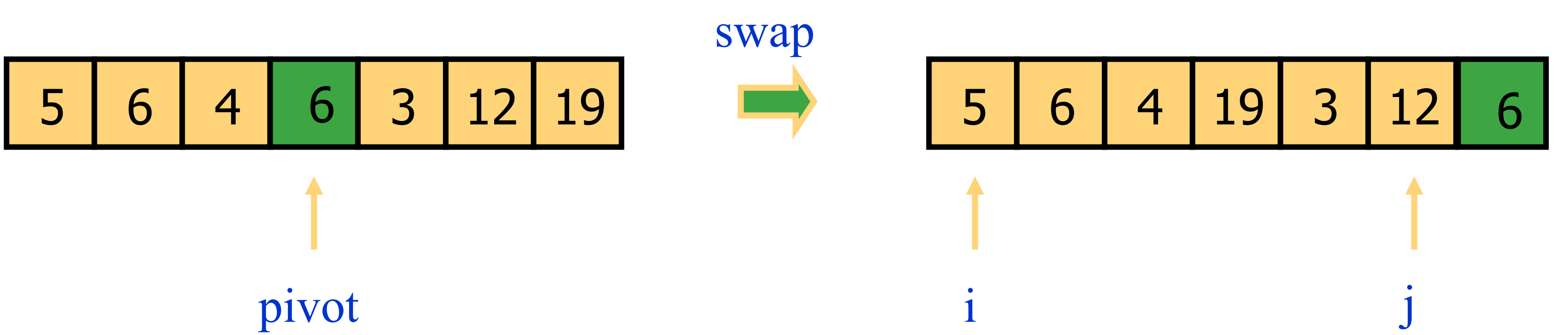
```
Quicksort (A, p, r) {
    if (p < r) {
        q = Partition (A, p, r) //q is the position of the
    pivot element
        Quicksort (A, p, q-1)
        Quicksort (A, q+1, r)
    }
}
```

Partitioning

- Partitioning
  - Key step of quicksort algorithm
  - Goal: given the picked pivot, partition the remaining elements into two smaller sets
  - Many ways to implement
  - Even the slightest deviations may cause surprisingly bad results.
- We will learn an easy and efficient partitioning strategy here.
- How to pick a pivot will be discussed later

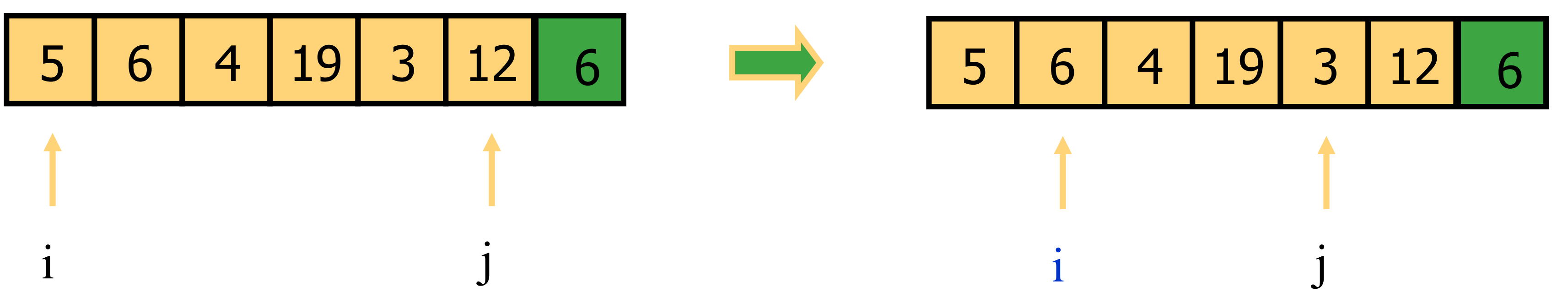
Partitioning Strategy

- Want to partition an array A[left .. right]
- First, get the pivot element out of the way by swapping it with the last element. (Swap pivot and A[right])
- Let i start at the first element and j start at the next-to-last element (i = left, j = right – 1)



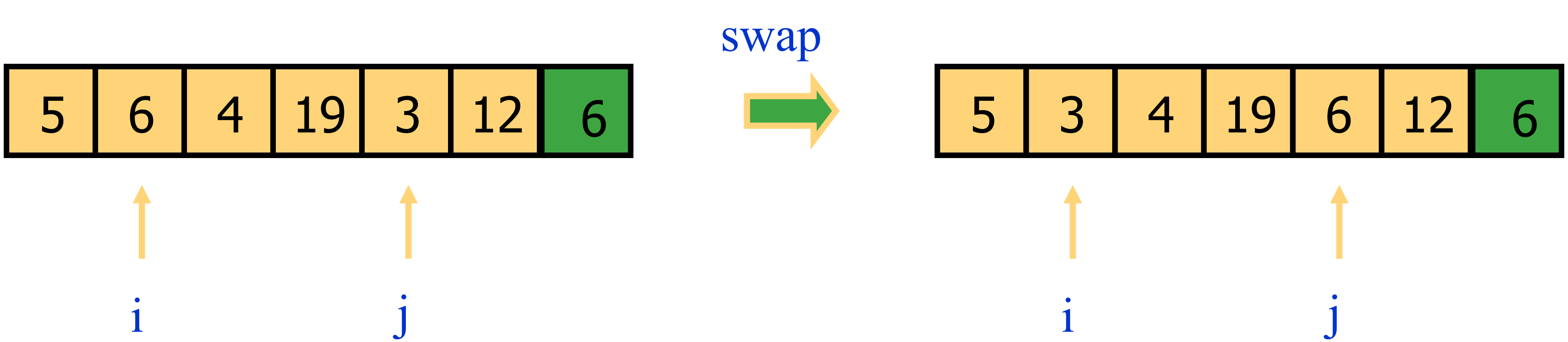
Partitioning Strategy

- Want to have
  - A[p] <= pivot, for p < i
  - A[p] >= pivot, for p > j
- When i < j
  - Move i right, skipping over elements smaller than the pivot
  - Move j left, skipping over elements greater than the pivot
  - When both i and j have stopped
    - A[i] >= pivot
    - A[j] <= pivot



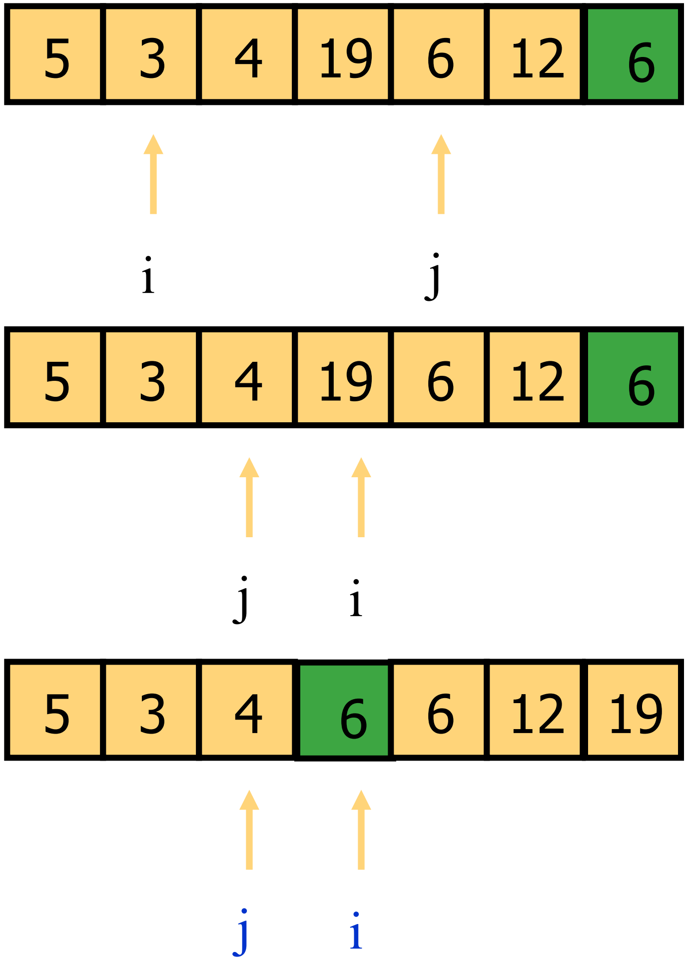
Partitioning Strategy

- When i and j have stopped and i is to the left of j
  - Swap A[i] and A[j]
    - The large element is pushed to the right and the small element is pushed to the left
  - After swapping
    - A[i] <= pivot
    - A[j] >= pivot
  - Repeat the process until i and j cross



Partitioning Strategy

- When i and j have crossed
  - Swap A[i] and pivot
- Result:
  - A[p] <= pivot, for p < i
  - A[p] >= pivot, for p > i



Small arrays

- For very small arrays, quicksort does not perform as well as insertion sort
  - how small depends on many factors, such as the time spent making a recursive call, the compiler, etc
- Do not use quicksort recursively for small arrays
  - Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort

Picking the Pivot

- Use the first element as pivot
  - if the input is random, ok
  - if the input is presorted (or in reverse order)
    - all the elements go into S2 (or S1)
    - this happens consistently throughout the recursive calls
    - Results in $O(n^2)$ behavior (Analyze this case later)
- Choose the pivot randomly
  - generally safe
  - random number generation can be expensive

Picking the Pivot

- Use the median of the array
  - Partitioning always cuts the array into roughly half
  - An optimal quicksort (O(N log N))
  - However, hard to find the exact median
    - e.g., sort an array to pick the value in the middle

Pivot: median of three

- We will use median of three
  - Compare just three elements: the leftmost, rightmost and center
  - Swap these elements if necessary so that
    - A[left] = Smallest
    - A[right] = Largest

```
int center = ( left + right ) / 2;
if( a[ center ] < a[ left ] )
    swap( a[ left ], a[ center ] );
if( a[ right ] < a[ left ] )
    swap( a[ left ], a[ right ] );
if( a[ right ] < a[ center ] )
    swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );
```

Pivot: median of three

2	5	6	4	13	3	12	19	6

A[left] = 2, A[center] = 13,
A[right] = 6

2	5	6	4	6	3	12	19	13

Swap A[center] and A[right]

2	5	6	4	6	3	12	19	13

Choose A[center] as pivot

pivot

2	5	6	4	19	3	12	6	13

Swap pivot and A[right – 1]

pivot

Note we only need to partition A[left + 1, …, right – 2]. Why?

Main Quicksort Routine

```
if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right );

        // Begin partitioning
    int i = left, j = right - 1;
    for( ; ; )
    {
        while( a[ ++i ] < pivot ) { }
        while( pivot < a[ --j ] ) { }
        if( i < j )
            swap( a[ i ], a[ j ] );
        else
            break;
    }

    swap( a[ i ], a[ right - 1 ] );   // Restore pivot

    quicksort( a, left, i - 1 );      // Sort small elements
    quicksort( a, i + 1, right );     // Sort large elements
}
else  // Do an insertion sort on the subarray
    insertionSort( a, left, right );
```

Choose pivot

Partitioning

Recursion

For small arrays

Partitioning Part

- Works only if pivot is picked as median-of-three.
  - A[left] <= pivot and A[right] >= pivot
  - Thus, only need to partition A[left + 1, …, right – 2]
- j will not run past the end
  - because a[left] <= pivot
- i will not run past the end
  - because a[right-1] = pivot

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

Analysis

- Assumptions:
  - A random pivot (no median-of-three partitioning
  - No cutoff for small arrays
- Running time
  - pivot selection: constant time O(1)
  - partitioning: linear time O(N)
  - running time of the two recursive calls
- T(N)=T(i)+T(N-i-1)+cN where c is a constant
  - i: number of elements in S1

Worst-Case Analysis

- What will be the worst case?
  - The pivot is the smallest element, all the time
  - Partition is always unbalanced

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

Best-case Analysis

- What will be the best case?
  - Partition is perfectly balanced.
  - Pivot is always in the middle (median of the array)

$$
\begin{aligned}
T(N) &= 2T(N/2) + cN \\
\frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + c \\
\frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + c \\
\frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + c \\
&\vdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + c \\
\frac{T(N)}{N} &= \frac{T(1)}{1} + c\log N \\
T(N) &= cN\log N + N = O(N\log N)
\end{aligned}
$$

Average-Case Analysis

- Assume
  - Each of the sizes for S1 is equally likely
- This assumption is valid for our pivoting (median-of-three) and partitioning strategy
- On average, the running time is O(N log N)

Exercises: P405-1, 2, 5

Mergesort

Based on divide-and-conquer strategy

- Divide the list into two smaller lists of about equal sizes
- Sort each smaller list *recursively*
- Merge the two sorted lists to get one sorted list

How do we divide the list? How much time needed?

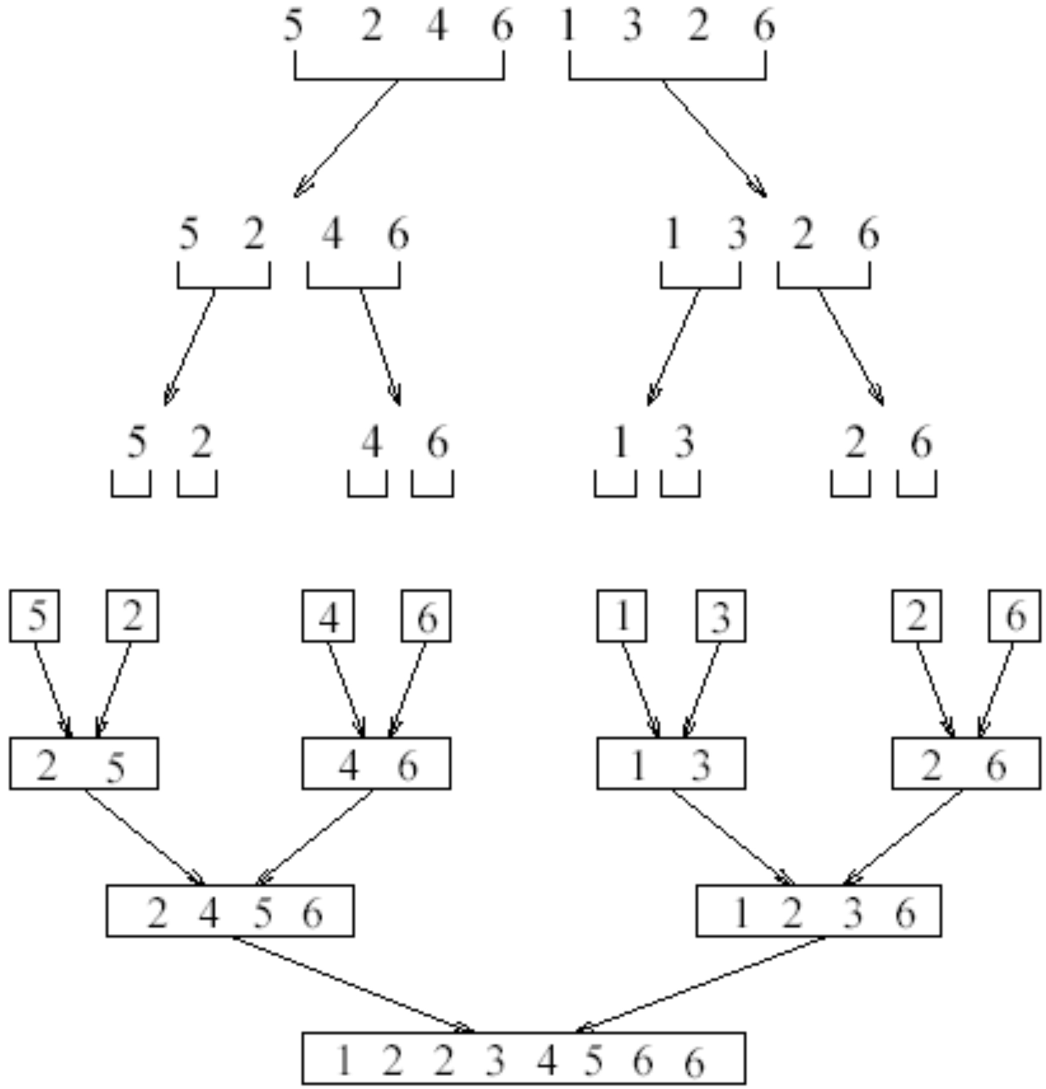
How do we merge the two sorted lists? How much time needed?

Dividing

- If the input list is a linked list, dividing takes $\Theta(N)$ time
  - We scan the linked list, stop at the $\lfloor N/2 \rfloor$ th entry and cut the link
- If the input list is an array A[0..N-1]: dividing takes $O(1)$ time
  - we can represent a sublist by two integers `left` and `right`: to divide `A[left..Right]`, we compute `center=(left+right)/2` and obtain `A[left..Center]` and `A[center+1..Right]`

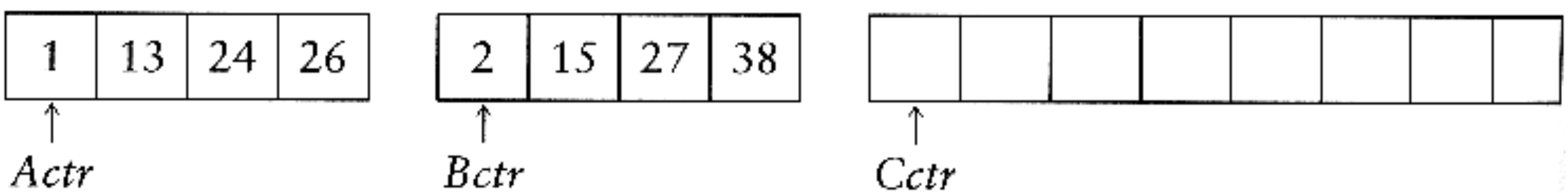
Mergesort

```
void mergesort(vector<int> & A, int left, int right)
{
    if (left < right) {
        int center = (left + right)/2;
        mergesort(A,left,center);
        mergesort(A,center+1,right);
        merge(A,left,center+1,right);
    }
}
```


5 2 4 6 1 3 2 6
5 2 4 6
1 3 2 6
5 2
4 6
1 3
2 6
5
2
4
6
1
3
2
6
2 5
4 6
1 3
2 6
2 4 5 6
1 2 3 6
1 2 2 3 4 5 6 6

How to merge?

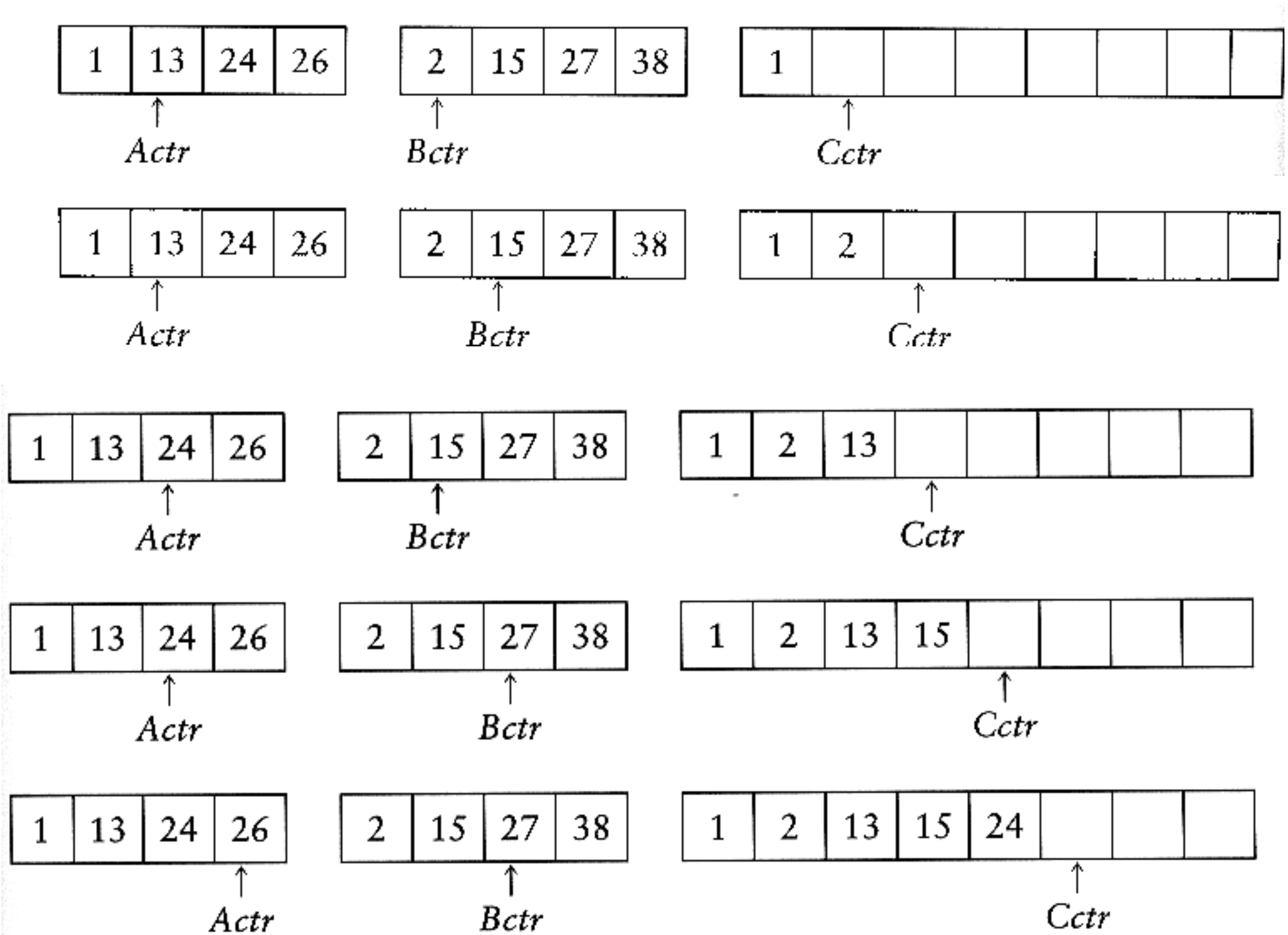
- Input: two sorted array A and B
- Output: an output sorted array C
- Three counters: Actr, Bctr, and Cctr
  - initially set to the beginning of their respective arrays



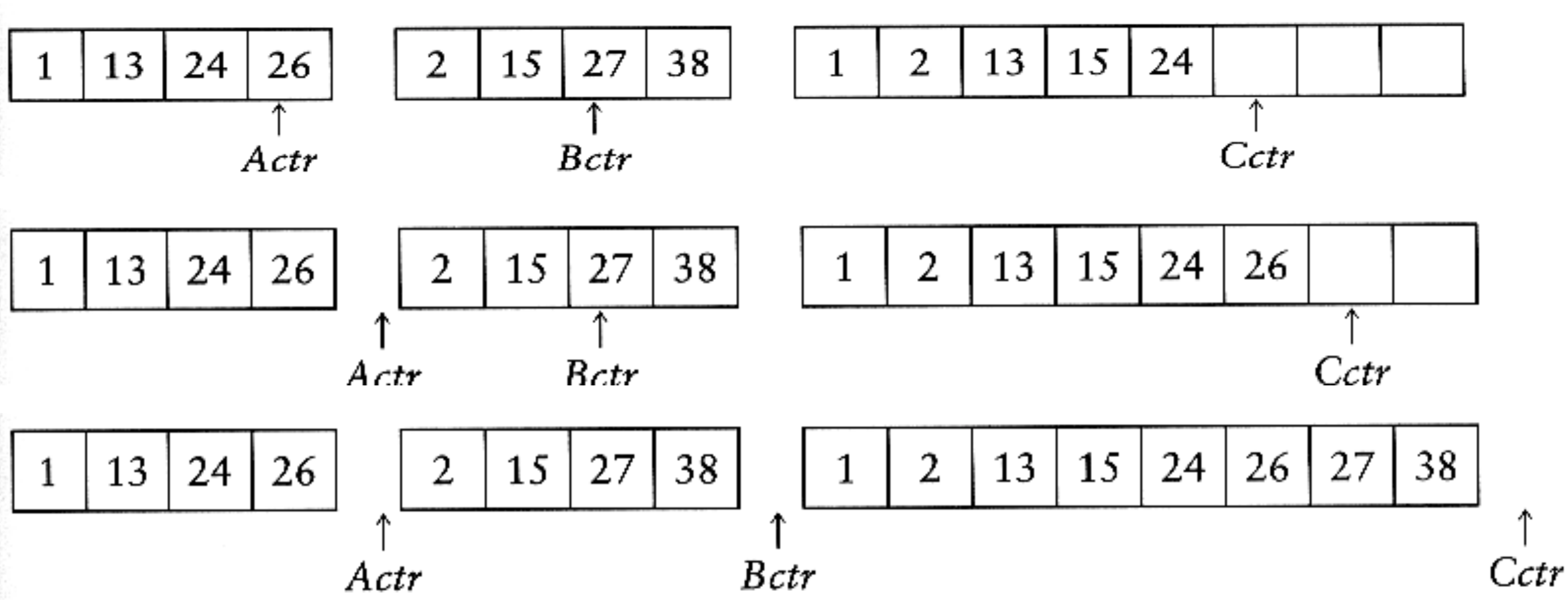
(1) The smaller of A[Actr] and B[Bctr] is copied to the next entry in C, and the appropriate counters are advanced

(2) When either input list is exhausted, the remainder of the other list is copied to C

Example: Merge



Example: Merge...



- Running time analysis:
  - Clearly, `merge` takes O(m1 + m2) where m1 and m2 are the sizes of the two sublists.
- Space requirement:
  - merging two sorted lists requires linear extra memory
  - additional work to copy to the temporary array and back

Algorithm *merge*(A, p, q, r)

Input: Subarrays $A[p..l]$ and $A[q..r]$ s.t. $p \leq l = q - 1 < r$.

Output: $A[p..r]$ is sorted.

(∗ T is a temporary array. ∗)

$k = p$; $i = 0$; $l = q - 1$;
while $p \leq l$ and $q \leq r$
    do if $A[p] \leq A[q]$
        then $T[i] = A[p]$; $i = i + 1$; $p = p + 1$;
        else $T[i] = A[q]$; $i = i + 1$; $q = q + 1$;
while $p \leq l$
    do $T[i] = A[p]$; $i = i + 1$; $p = p + 1$;
while $q \leq r$
    do $T[i] = A[q]$; $i = i + 1$; $q = q + 1$;
for $i = k$ to r
    do $A[i] = T[i - k]$;

Analysis of mergesort

Let $T(N)$ denote the worst-case running time of mergesort to sort N numbers.

Assume that N is a power of 2.

- Divide step: $O(1)$ time
- Conquer step: $2\,T(N/2)$ time
- Combine step: $O(N)$ time

Recurrence equation:

$$T(1) = 1$$

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

Analysis: solving recurrence

$$T(N) = 2T(\frac{N}{2}) + N$$
$$= 2(2T(\frac{N}{4}) + \frac{N}{2}) + N$$
$$= 4T(\frac{N}{4}) + 2N$$
$$= 4(2T(\frac{N}{8}) + \frac{N}{4}) + 2N$$
$$= 8T(\frac{N}{8}) + 3N = \cdots$$
$$= 2^k T(\frac{N}{2^k}) + kN$$

Since $N=2^k$, we have $k=\log_2 n$

$$T(N) = 2^k T(\frac{N}{2^k}) + kN$$
$$= N + N \log N$$
$$= O(N \log N)$$

Comparing $n \log_{10} n$ and n^2

n	$n \log_{10} n$	n^2	Ratio
100	0.2K	10K	50
1000	3K	1M	333.33
2000	6.6K	4M	606
3000	10.4K	9M	863
4000	14.4K	16M	1110
5000	18.5K	25M	1352
6000	22.7K	36M	1588
7000	26.9K	49M	1820
8000	31.2K	64M	2050

An experiment

- Code from textbook (using template)
- Unix `time` utility

n	Isort (secs)	Msort (secs)	Ratio
100	0.01	0.01	1
1000	0.18	0.01	18
2000	0.76	0.04	19
3000	1.67	0.05	33.4
4000	2.90	0.07	41
5000	4.66	0.09	52
6000	6.75	0.10	67.5
7000	9.39	0.14	67
8000	11.93	0.14	85

Iterative merge sort

- At the beginning, interpret the input as n sorted sublists, each of size 1
- These lists are merged by pairs to obtain n/2 lists, each of length 2 (if n is odd, then one list is of length 1)
- These n/2 lists are then merged by pairs
- Repeat until only one list is get.

49 38 65 97 76 13 27

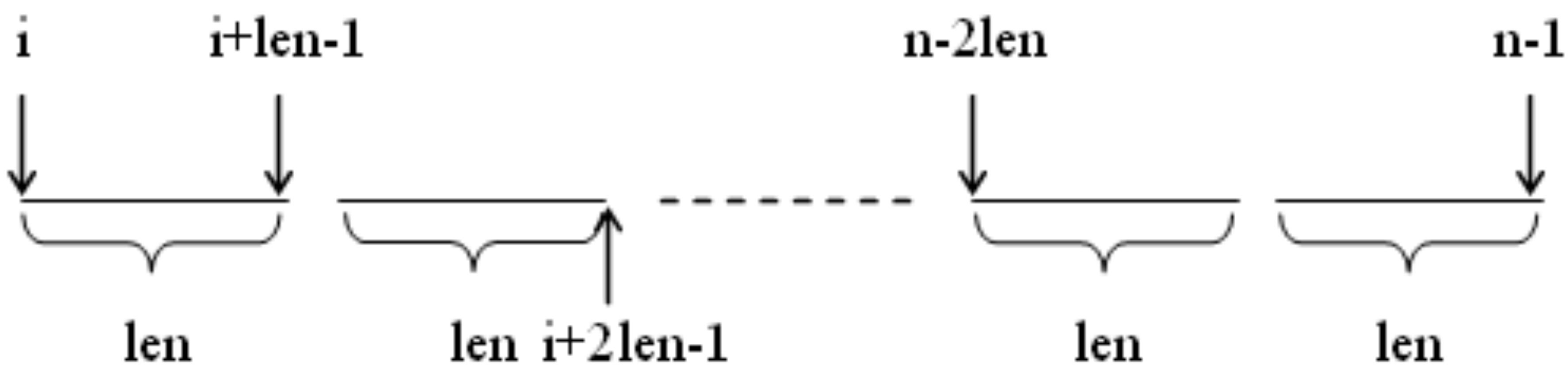
Pass 1 [49] [38] [65] [97] [76] [13] [27]

Pass 2 [38 49] [65 97] [13 76] [27]

Pass 3 [38 49 65 97] [13 27 76]

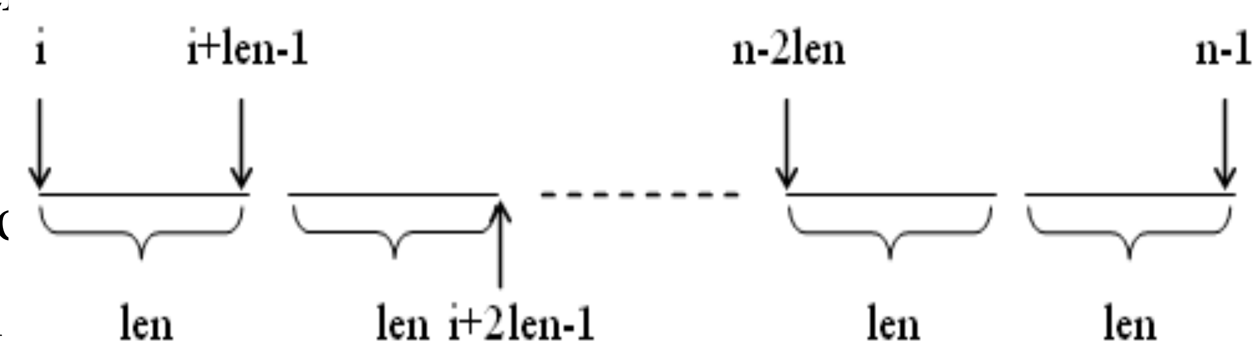
[13 27 38 49 65 76 97]

- ✓ Multiple scans of input file
- ✓ Given a len-sorted input file
  - After a scan-merge , (2len)-sorted file obtained



Problems

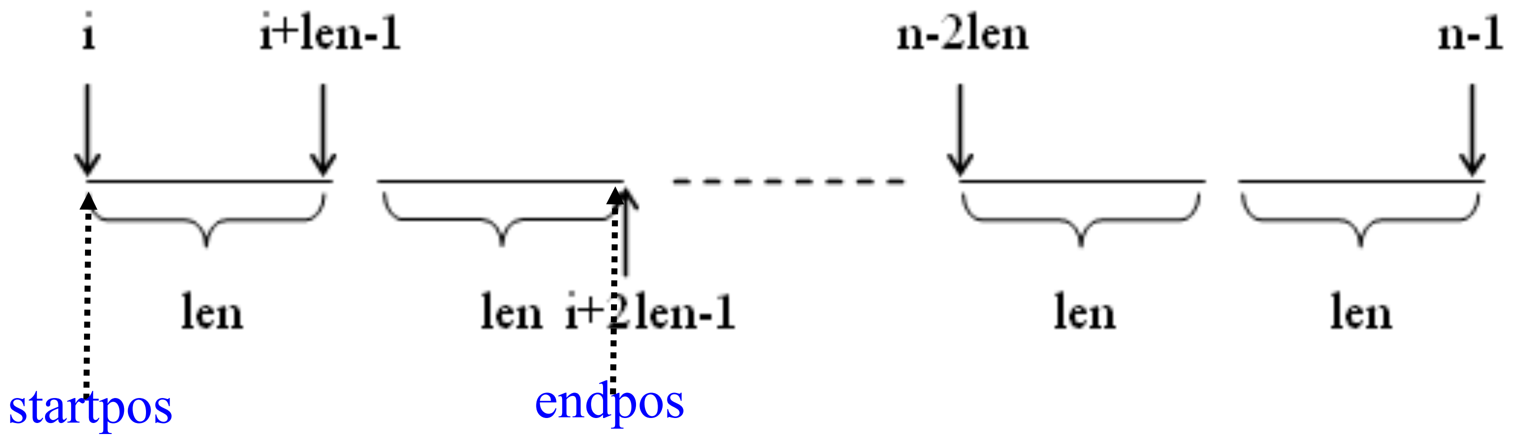
- Two-way merge algorithm
- Grouping the input file to do one pass merge
  - How?
    - Length c
    - If 2×len
      merged
    - Or, do something else



- Determine passes
  - How?
    - Length of sub-sequence : 1→n

2-way merge

- ✓Input：list, startpos, len, endpos



2-way merge

```
template <class KeyType>
void merge(Element<Type> *initList, Element<Type>
 *mergedList, const int startpos, const int len, const int endpos)
 {
  for (int i1 = start , i2 = i+len, iResult = start ;
          i1 <= i+len-1 && i2 <= endpos;
          iResult++)
    if ( initList[i1].getKey ( ) <= initList[i2].getKey ( )) {
      mergedList[iResult] = initList[i1];
      i1++;
    }
    else {
      mergedList[iResult] = initList[i2];
      i2++;
    }
```

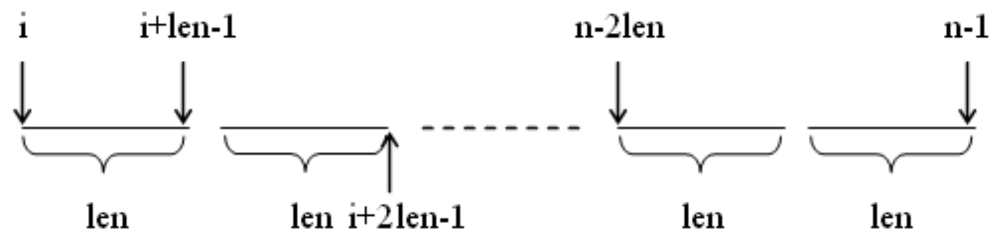
```
  if (i1 > i+len-1)
    for (int t = i2; t <= endpos; t++)
      mergedList[iResult+t-i2] = initList[t];
  else
    for (int t = i1; t <= i+len-1; t++)
      mergedList[iResult+t-i1] = initList[t];
}
```

Group & Merge

```
template <class KeyType>
void MergePass(Element<KeyType> *initList,
 Element<KeyType> *resultList, const int n, const int len) {

  for (int i = 0;
        i <= n – 2 * len;
        i += 2 * len)
    merge(initList, resultList, i, len, i+2 * len–1);

  if (i+len–1 < n–1)
    merge(initList, resultList, i, len, n–1);
  else
    for (int t = i; t <= n–1; t++)
        resultList[t] = initList[t];
}
```



Passes determining & mergesort

```
template <class KeyType>
void MergeSort(Element<KeyType> *list, const int n) {
  Element<KeyType> *tempList = new Element<KeyType> [n];
  for (int l = 1;
          l < n;
          l* = 2) {
    MergePass(list, tempList, n, l);
    l*=2;
    MergePass(tempList, list, n, l);
  }
  delete [ ] tempList;
}
```

Analysis:

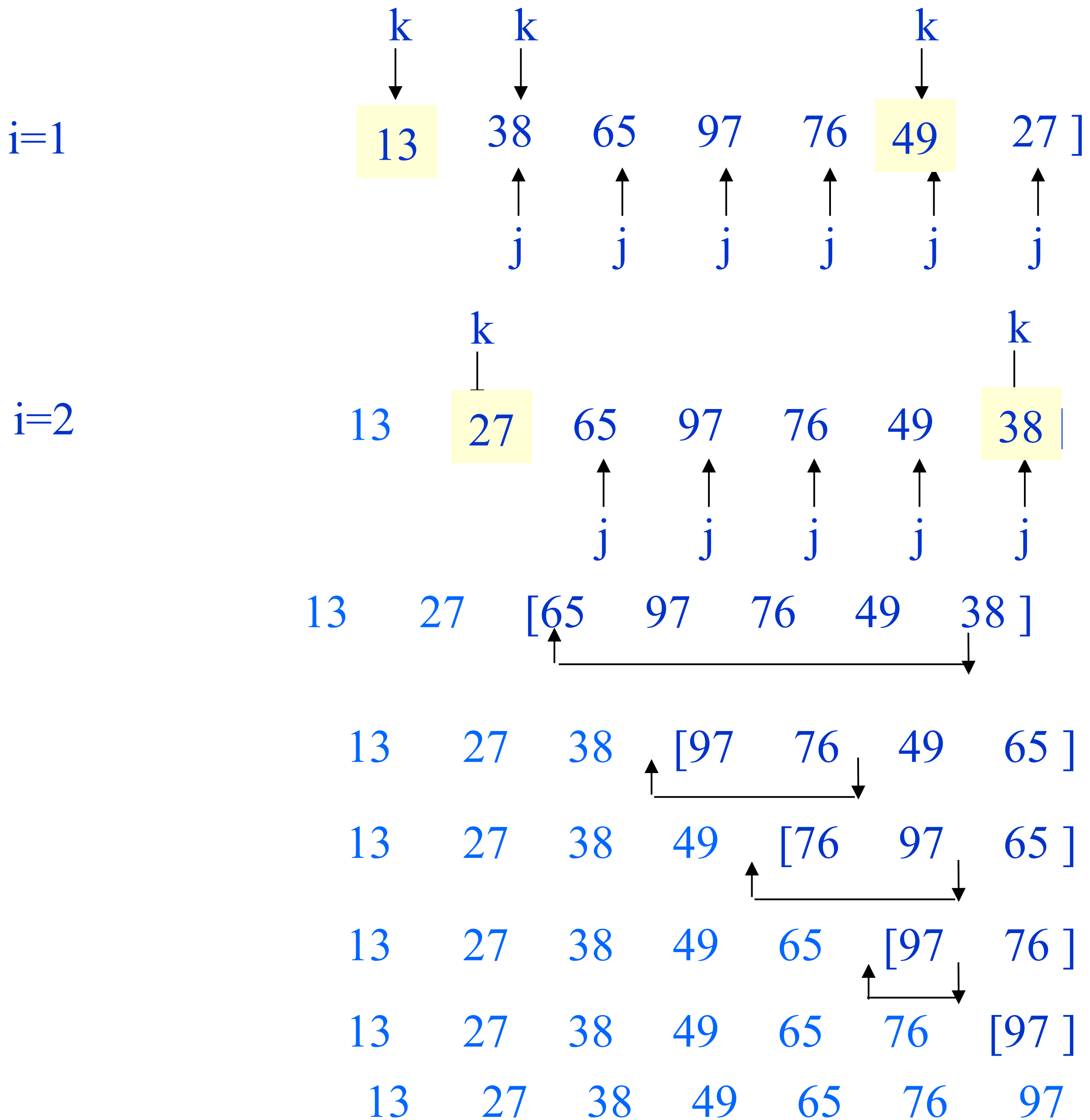
- Scan of input file
  - First pass: length of 1
  - Second pass: length of 2
  - i-th pass: length of 2^i
- How many passes?
  - $\lceil \log_2 n \rceil$
- Each pass: O(n)
- Time complexity of mergesort

  O(n log n)

Exercises: P412-1

Select sort

- Naïve algorithm
- Basic idea
  - Select the smallest item by n-1 comparisons
    - Exchange it with the first item
  - Select the smallest item of remaining n-1 items by n-2 comparisons
    - Exchange it with the second item
  - ……
  - Repeat n-1 times



i=1
k k k
13 38 65 97 76 49 27]
j j j j j j
i=2
k k
13 27 65 97 76 49 38]
j j j j j
13 27 [65 97 76 49 38]
13 27 38 [97 76 49 65]
13 27 38 49 [76 97 65]
13 27 38 49 65 [97 76]
13 27 38 49 65 76 [97]
13 27 38 49 65 76 97

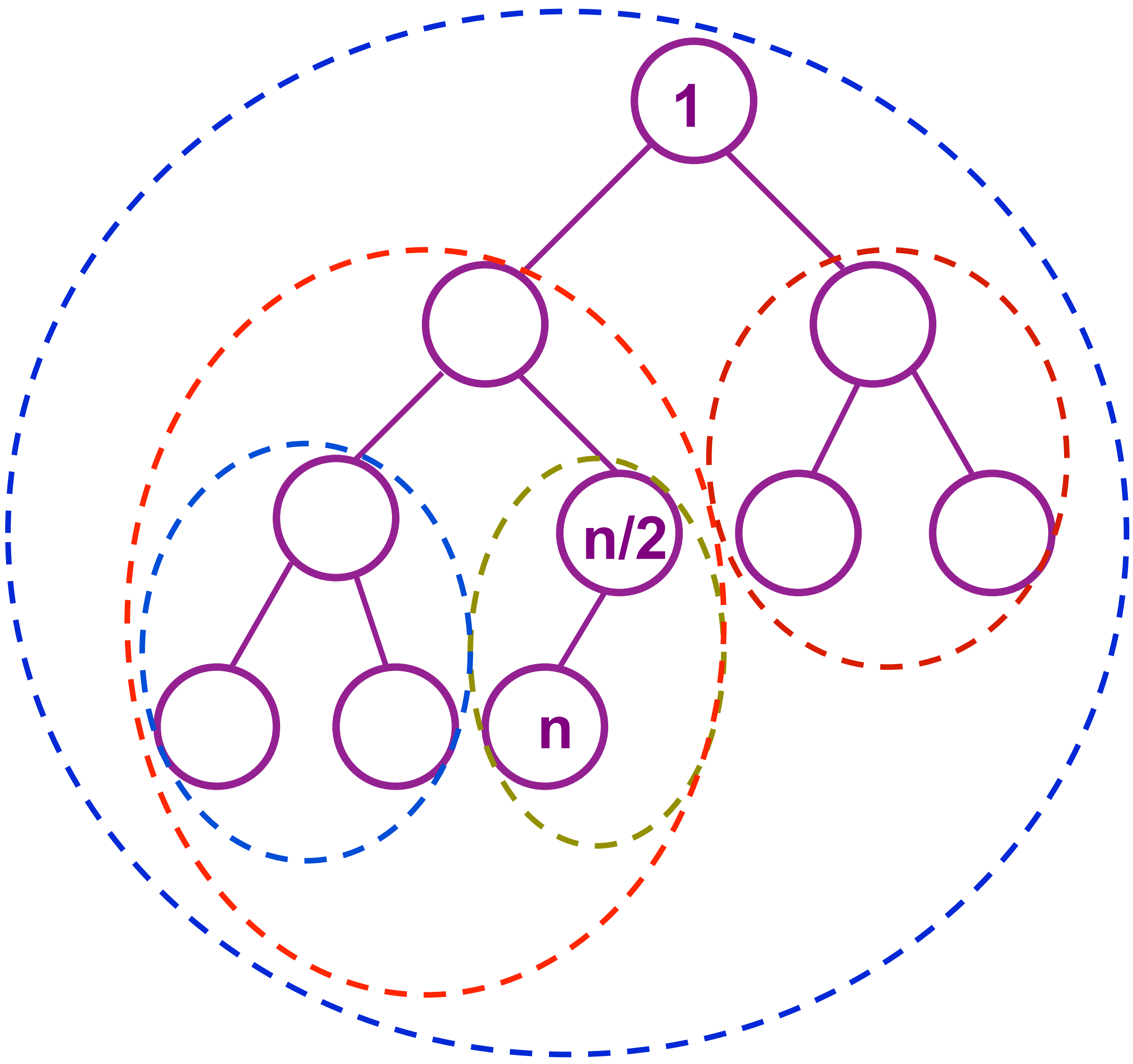
```
void smp_selesort(JD r[],int n) {
  int i,j,k;
  JD x;
  for(i=1;i<n;i++) {
    k=i;
    for(j=i+1;j<=n;j++)
      if(r[j].key<r[k].key)
          k=j;
    if(i!=k) {
      x=r[i];
      r[i]=r[k];
      r[k]=x;
    }
  }
}
```

- Analysis
  - $O(n^2)$

Heap sort

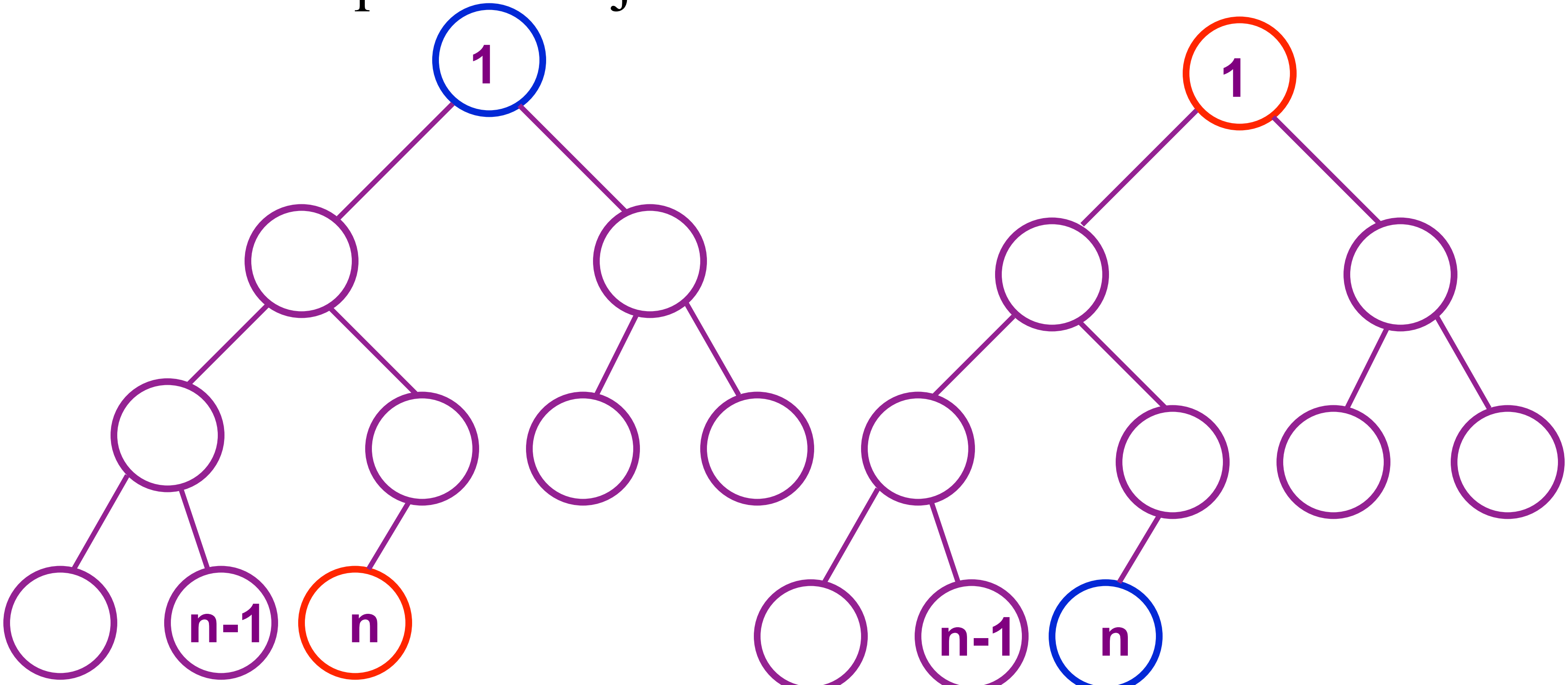
- Initialize a heap
- Output min/max
- Adjust the heap
- Repeat output/adjust until …

Heap initialization

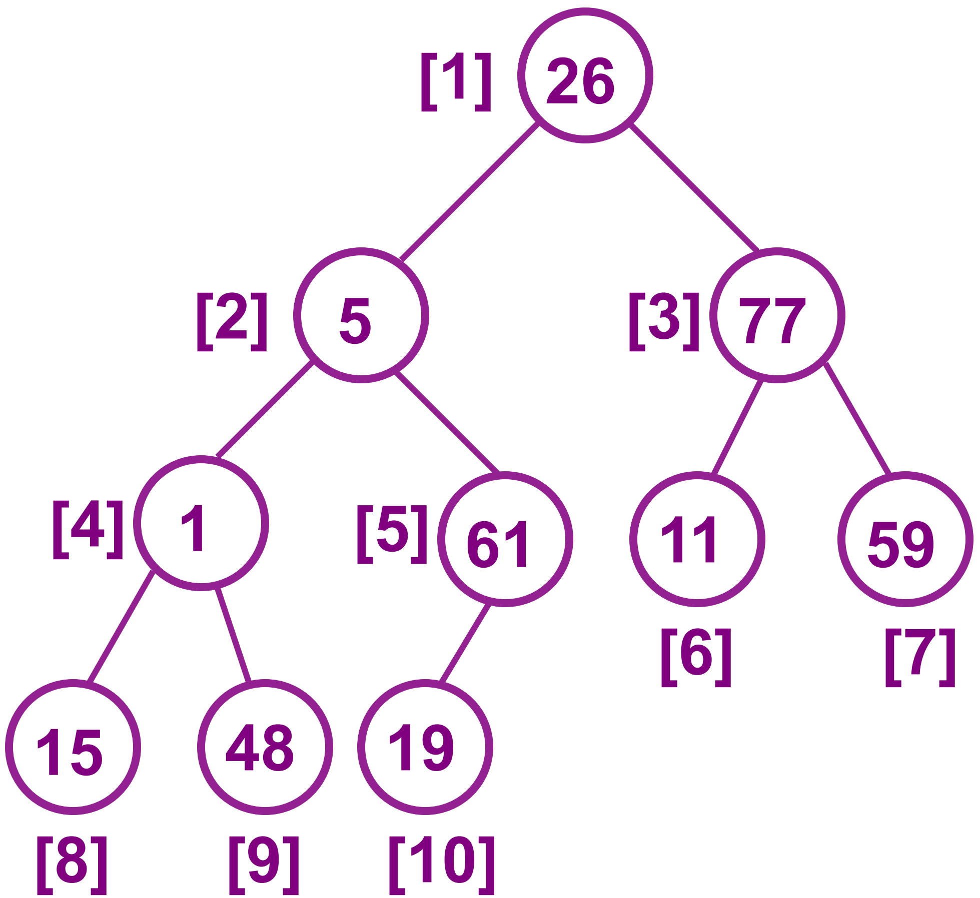


Heap sort

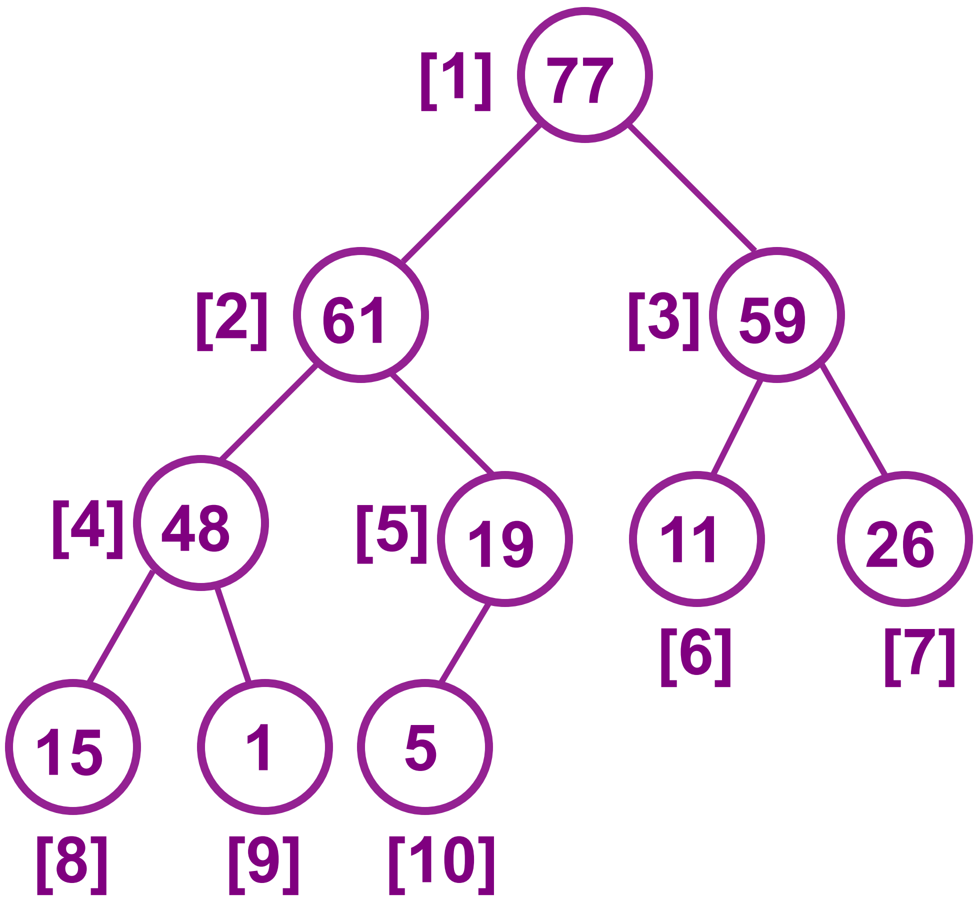
- With no extra space
  - Swap and readjust



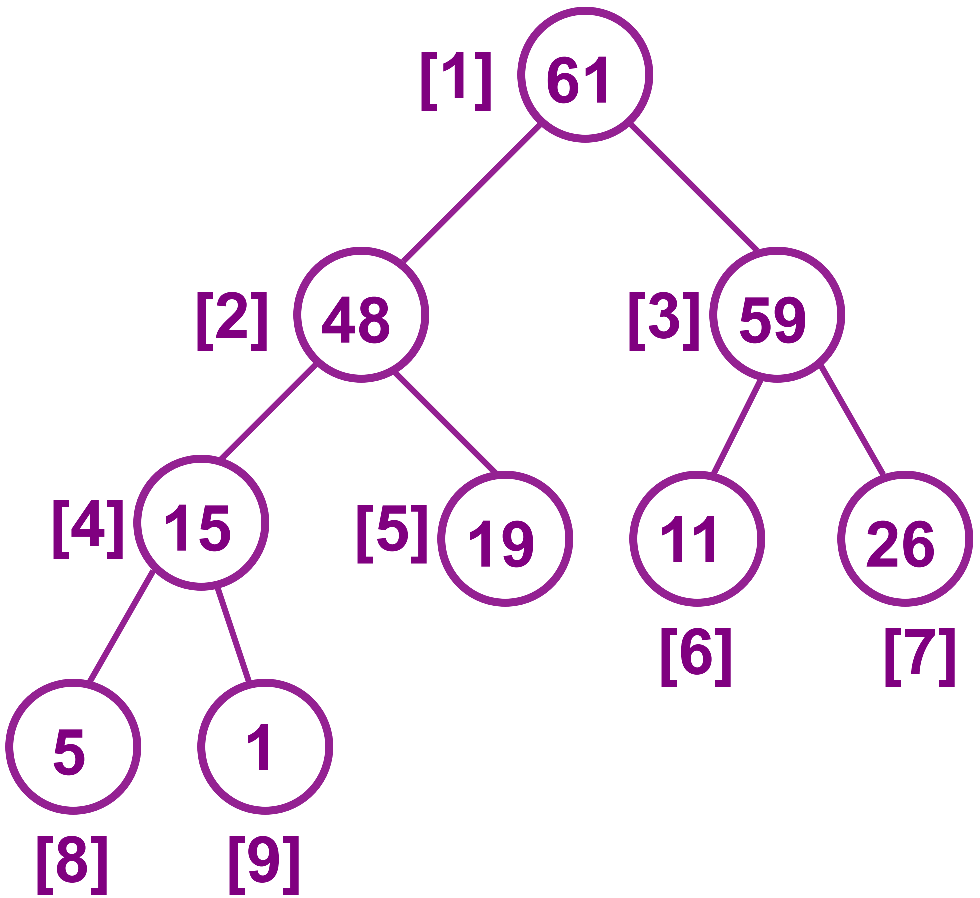
```
template <class T>
void HeapSort (T *list, const int n)
{ // Sort a[1:n] into nondeceasing order.
   for (int i=n/2; i>=1; i--)   // convert list into a heap
      Adjust(a, i, n);
   for (i=n-1; i>=1; i--)        // sort
   {
      swap(a[1], a[i+1];    // swap first and last
      Adjust(a, 1, i);        // recreate heap
      }
}
```



(a) Input list

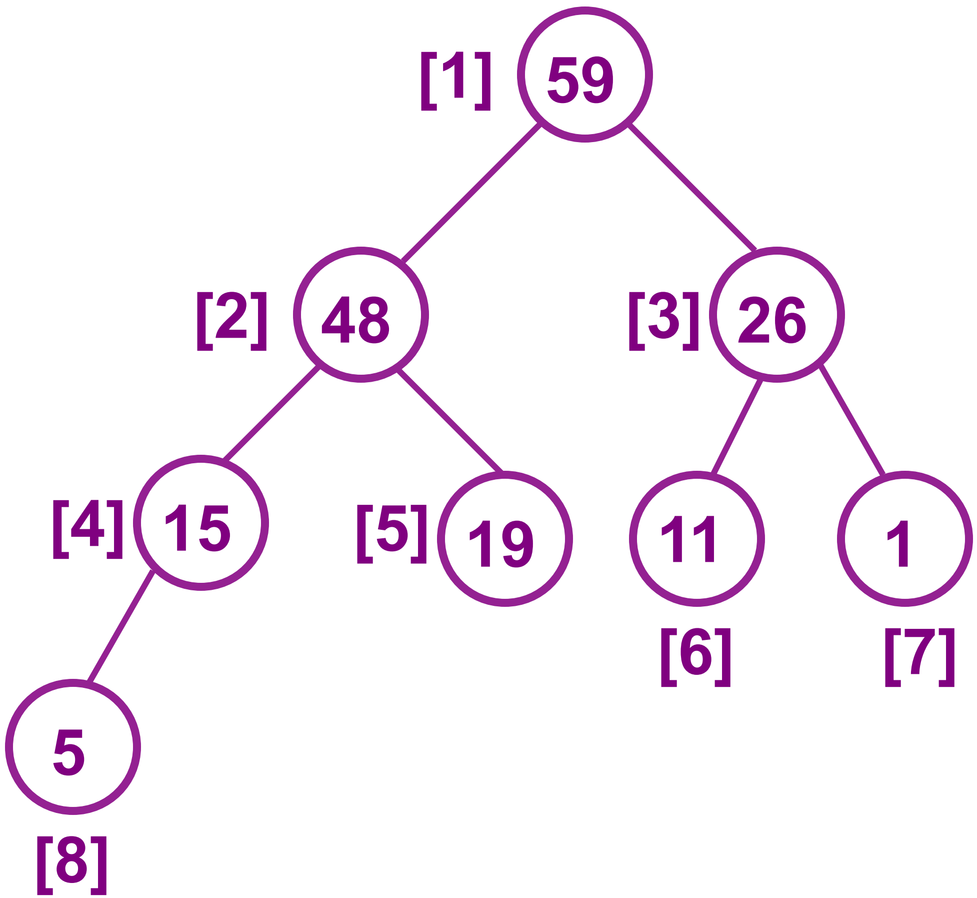


(b) Initial heap



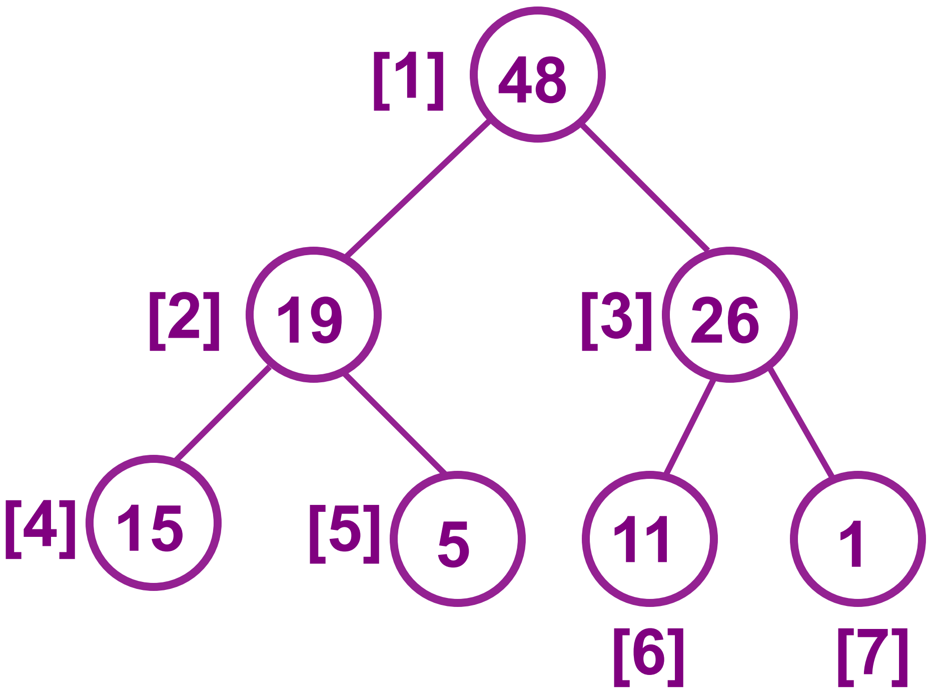
(c) Heap size=9

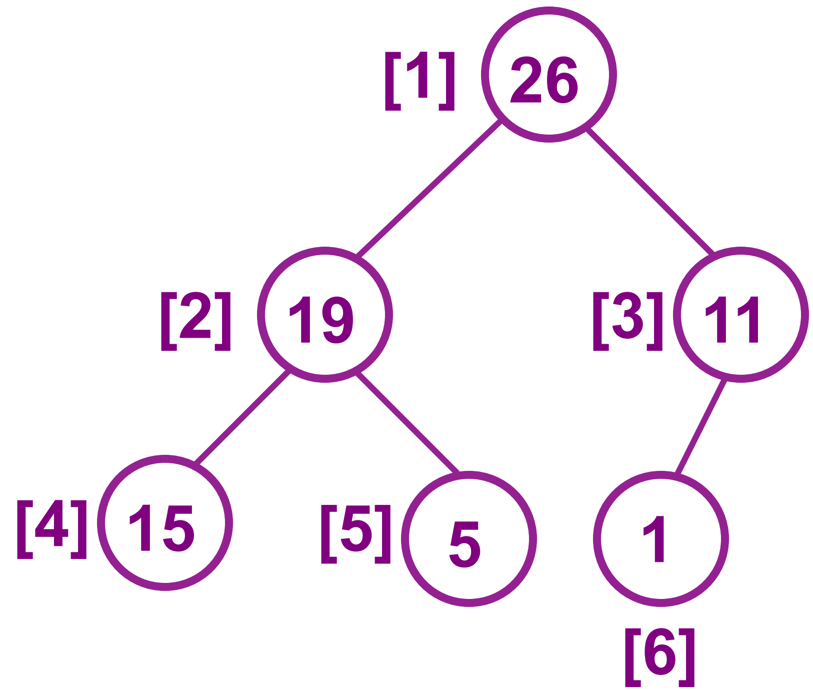
Sorted=[77]



(d) Heap size=8

Sorted=[61, 77]



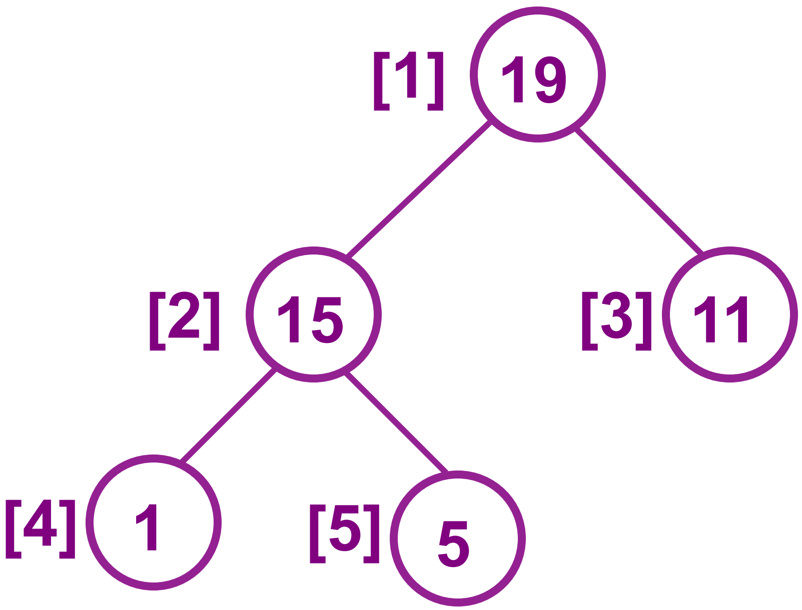


(e) Heap size=7

Sorted=[59, 61, 77]

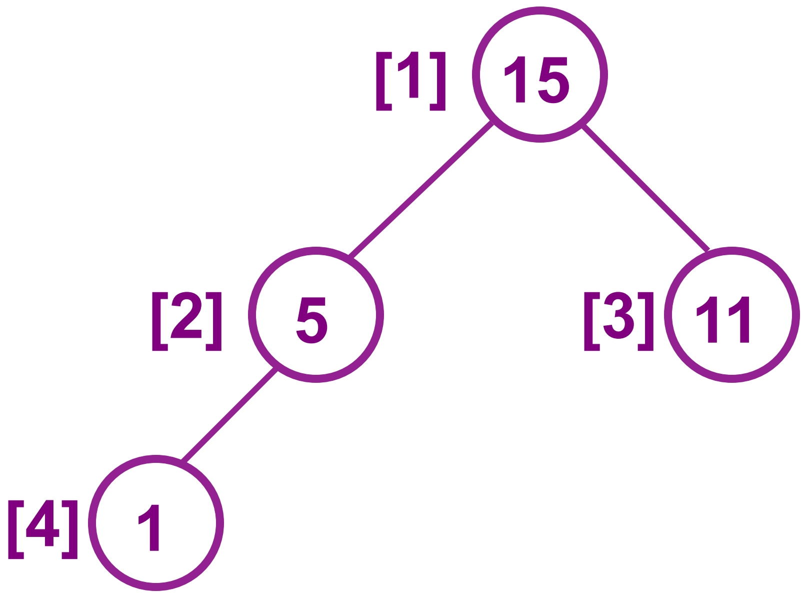
(f) Heap size=6

Sorted=[48, 59, 61, 77]



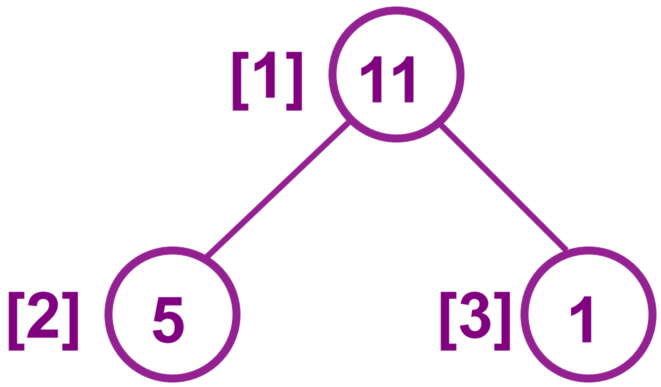
(g) Heap size=5

[26, 48, 59, 61, 77]



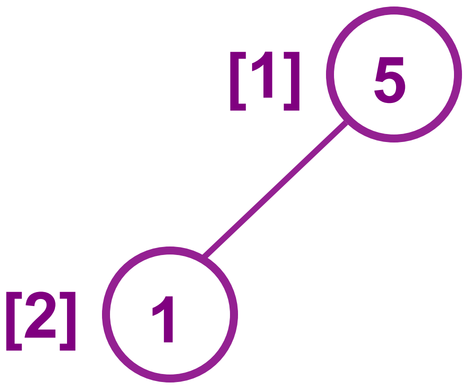
(h) Heap size=4

[19, 26, 48, 59, 61, 77]



(i) Heap size=3

[15, 19, 26, 48, 59, 61, 77]



(j) Heap size=2

[11, 15, 19, 26, 48, 59, 61, 77]

(j) Heap size=1

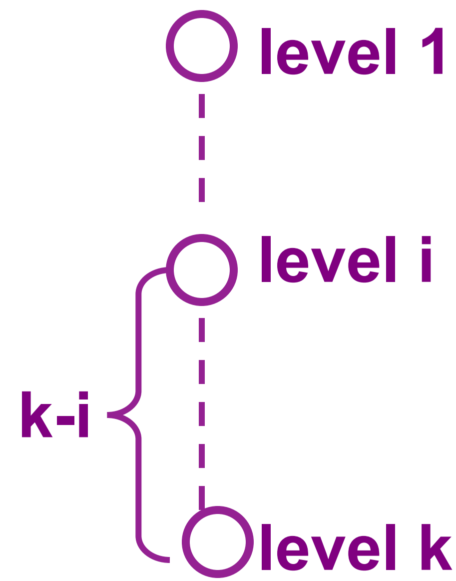
[5, 11, 15, 19, 26, 48, 59, 61, 77]

Analysis of HeapSort:

- suppose $2^{k-1} \le n < 2^k$, the tree has k levels.
- the number of nodes on level i $\le 2^{i-1}$.
- in the first loop, Adjust is called once for each node that has a child, hence the time is no more than

$$\sum_{1\le i\le k-1} 2^{i-1}(k-i) =$$

$$\sum_{1\le i\le k-1} 2^{k-i-1} i \le n \sum_{1\le i\le k-1} i/2^i < 2n = O(n)$$



- in the next loop, n-1 applications of Adjust are made with maximum depth $k = \lceil \log_2(n+1) \rceil$.

The total time: O(n log n).

Additional space: O(1).

Exercises: P416-1, 2

Radix Sort

- Extra information: every integer can be represented by at most k digits
  - $d_1 d_2 \ldots d_k$ where d_i are digits in base r
  - d_1: most significant digit
  - d_k: least significant digit

Radix sort

- ***Origin***: Herman Hollerith's card-sorting machine for the 1890 U.S. Census
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on ***least-significant digit first*** with auxiliary ***stable*** sort.

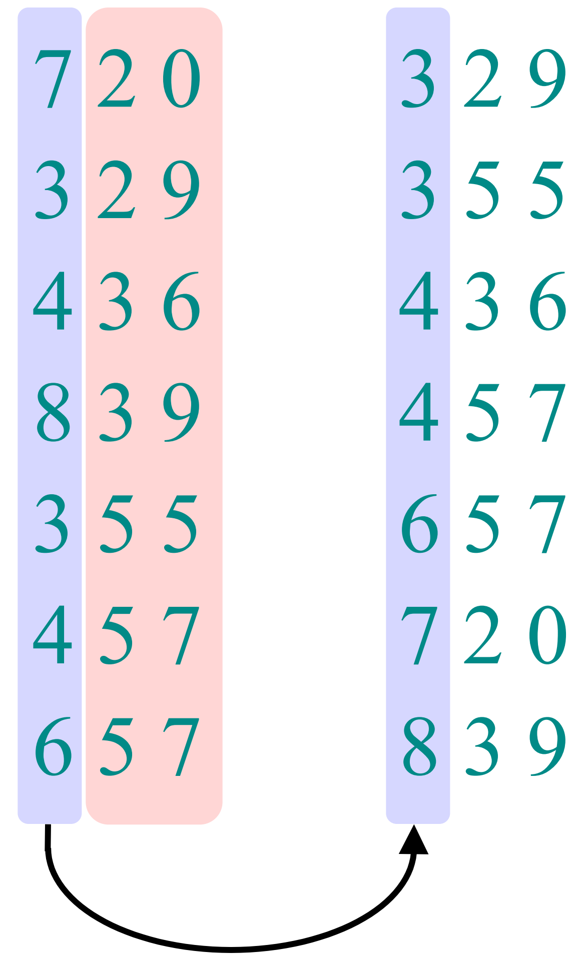
Operation of radix sort

3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Correctness of radix sort

Induction on digit position

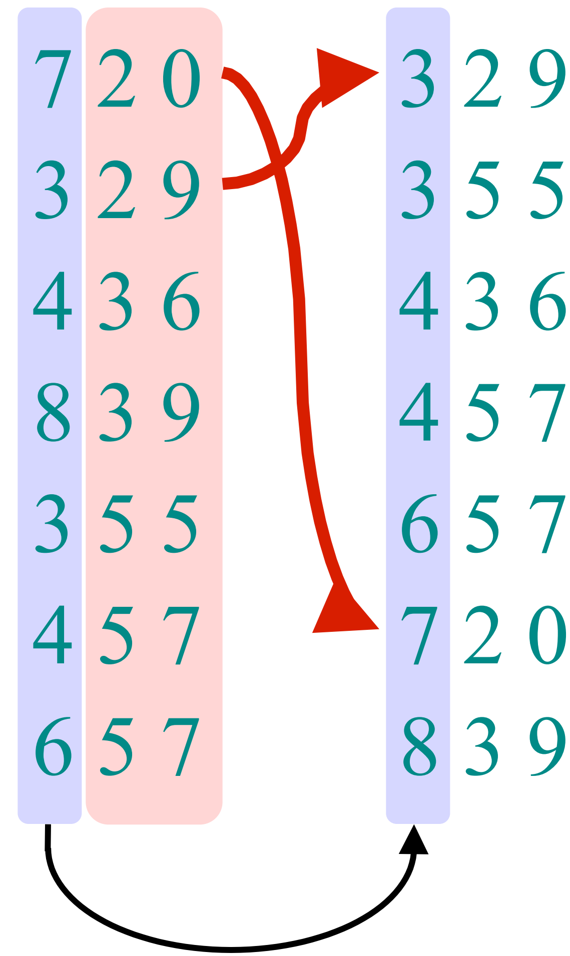
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t



Correctness of radix sort

Induction on digit position

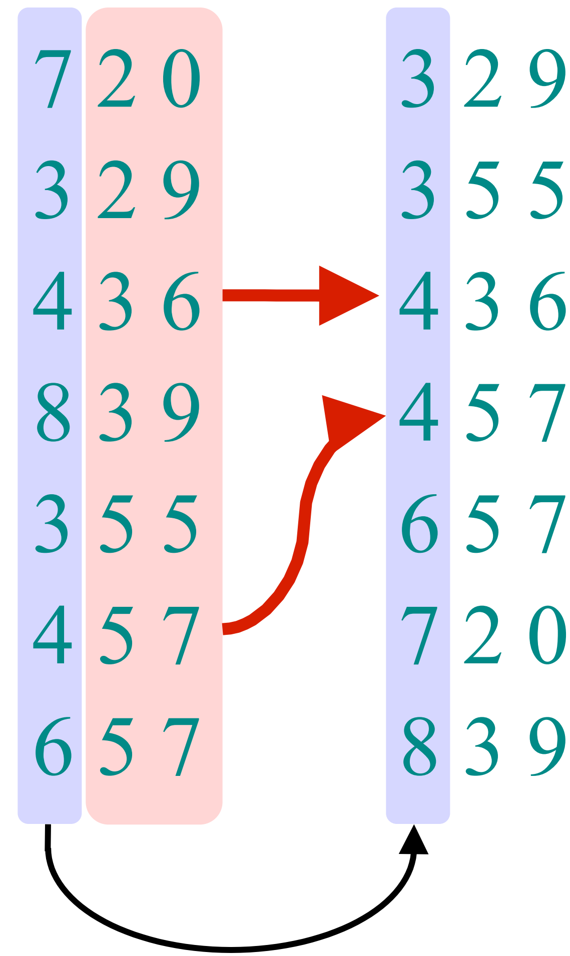
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.



Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input ⇒ correct order.



Radix Sort

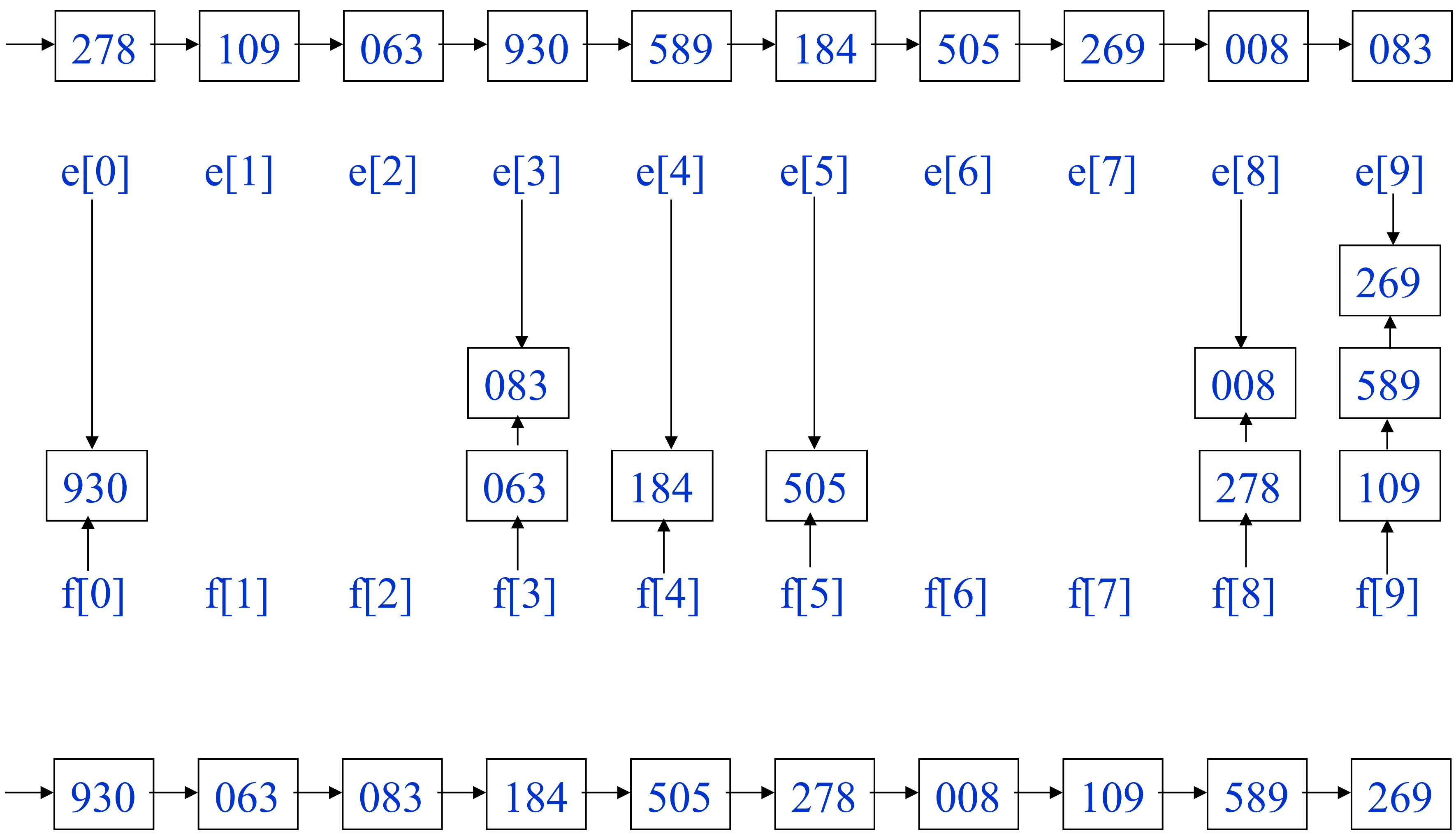
- Algorithm
  - sort by the least significant digit first

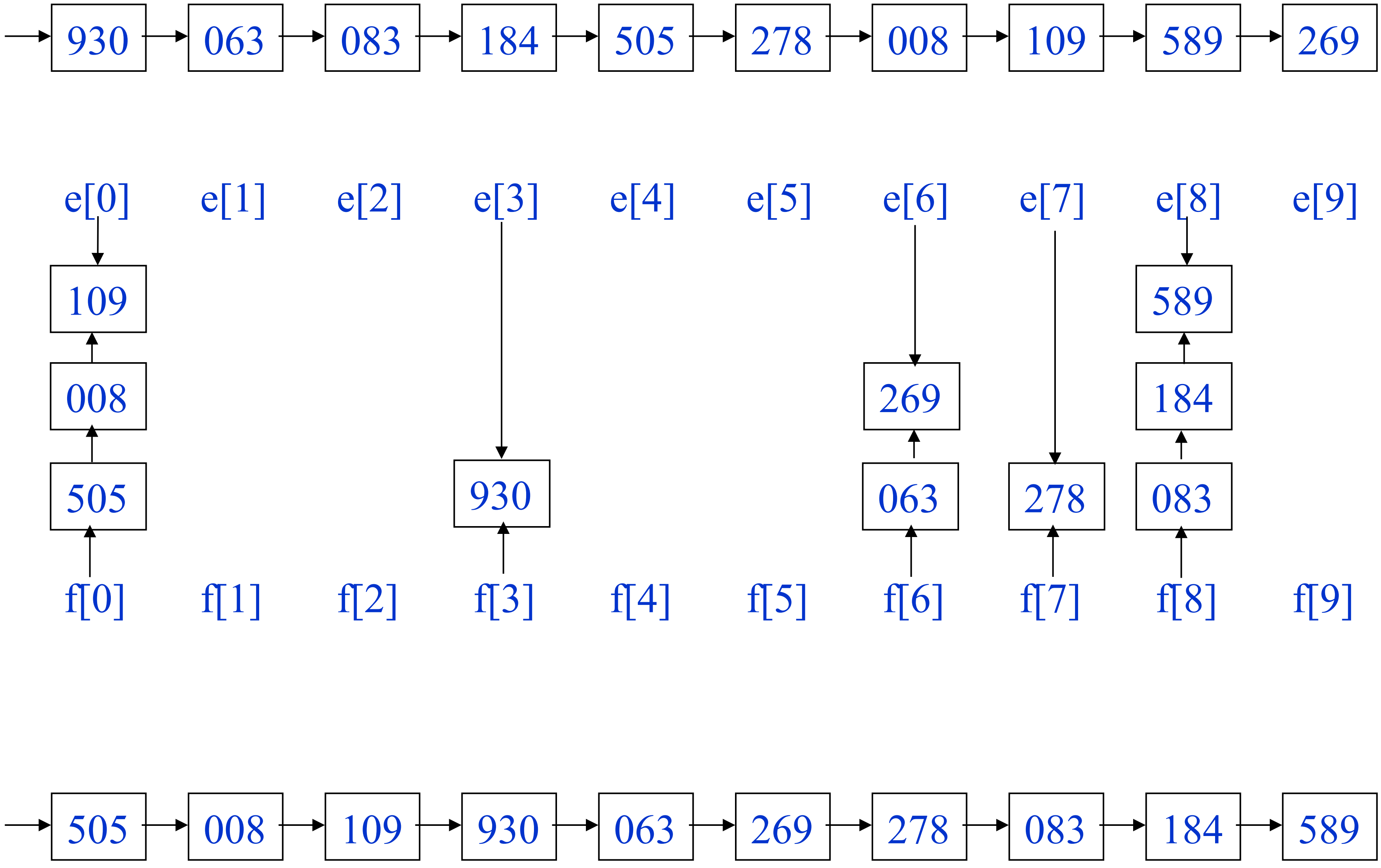
    => Numbers with the same digit go to same bin
  - reorder all the numbers: the numbers in bin 0 precede the numbers in bin 1, which precede the numbers in bin 2, and so on
  - sort by the next least significant digit
  - continue this process until the numbers have been sorted on all k digits

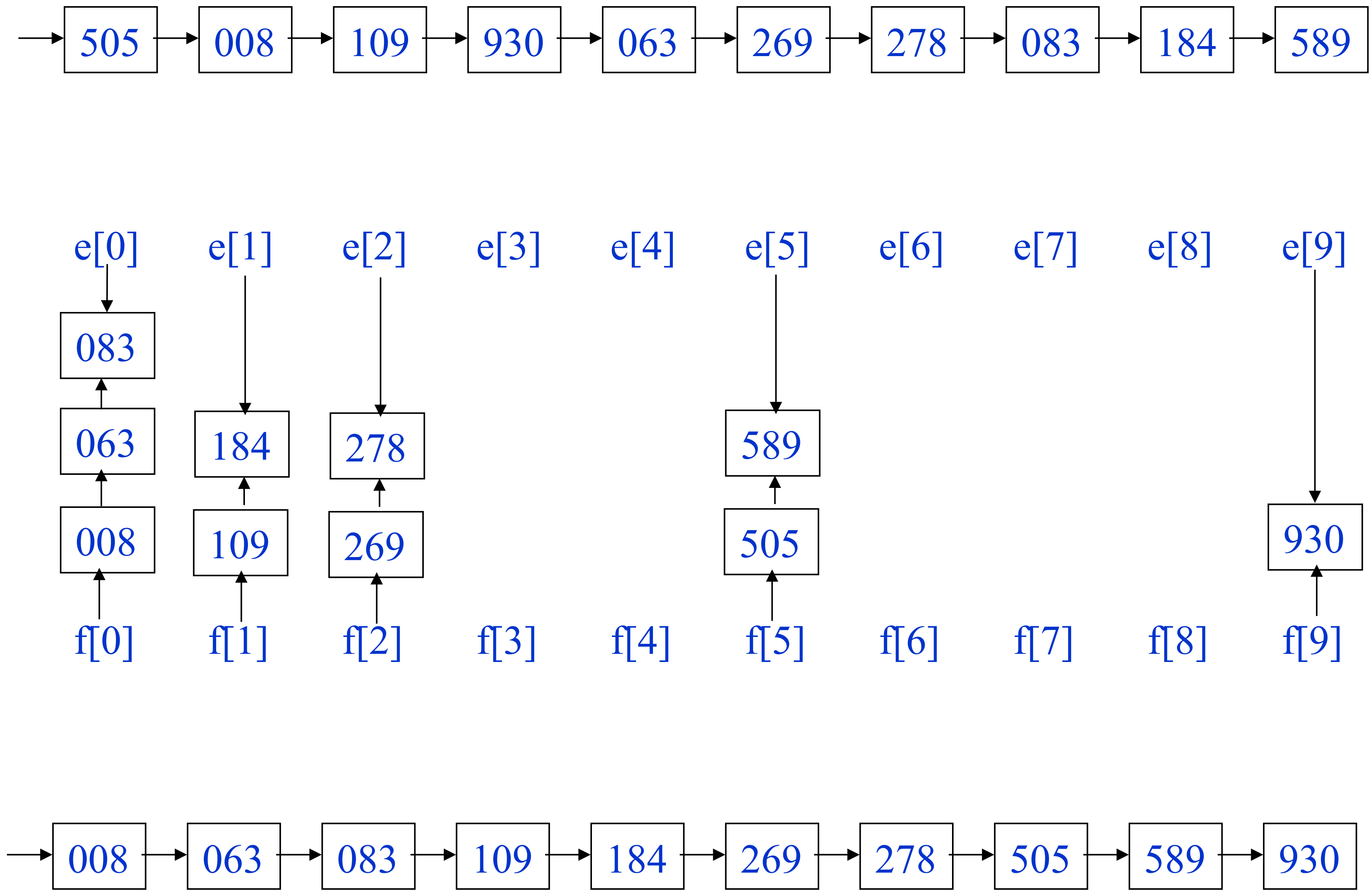
Algorithm *RadixSort*(A, n, d)

```
for 0 ≤ p ≤ 9                                   // base 10
    do Q[p] := empty queue;    // FIFO
D := 1;
for 1 ≤ k ≤ d                         // d times of counting sort
    do
        D := 10 * D;
        for 0 ≤ i < n   // scan A[i], put into correct slot
            do t := (A[i] mod D) div (D/10);
                enqueue(A[i], Q[t]);
        j := 0;
        for 0 ≤ p ≤ 9   // re-order back to original array
            do while Q[p] is not empty
                do A[j] := dequeue(Q[p]);
                    j := j + 1;
```

Starting:


278
109
063
930
589
184
505
269
008
083
e[0]
e[1]
e[2]
e[3]
e[4]
e[5]
e[6]
e[7]
e[8]
e[9]
269
083
008
589
930
063
184
505
278
109
f[0]
f[1]
f[2]
f[3]
f[4]
f[5]
f[6]
f[7]
f[8]
f[9]
930
063
083
184
505
278
008
109
589
269

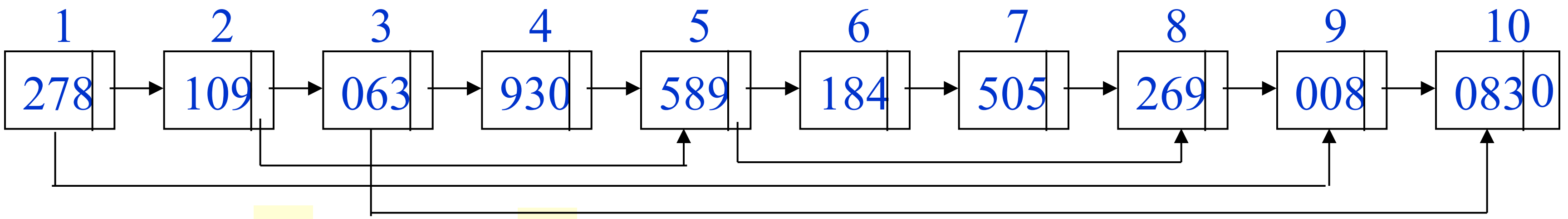

930
063
083
184
505
278
008
109
589
269
e[0]
e[1]
e[2]
e[3]
e[4]
e[5]
e[6]
e[7]
e[8]
e[9]
109
589
008
269
184
505
930
063
278
083
f[0]
f[1]
f[2]
f[3]
f[4]
f[5]
f[6]
f[7]
f[8]
f[9]
505
008
109
930
063
269
278
083
184
589



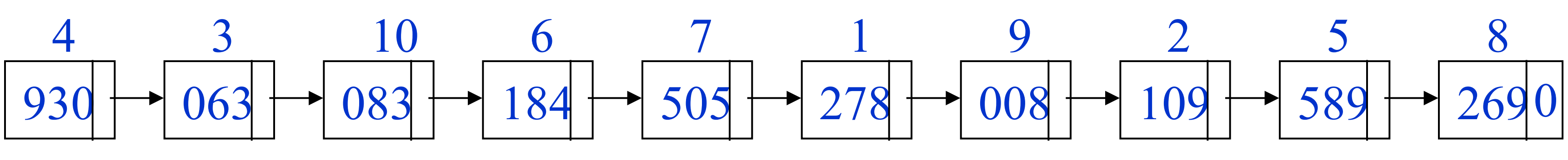
505
008
109
930
063
269
278
083
184
589
e[0]
e[1]
e[2]
e[3]
e[4]
e[5]
e[6]
e[7]
e[8]
e[9]
083
063
008
184
109
278
269
589
505
930
f[0]
f[1]
f[2]
f[3]
f[4]
f[5]
f[6]
f[7]
f[8]
f[9]
008
063
083
109
184
269
278
505
589
930

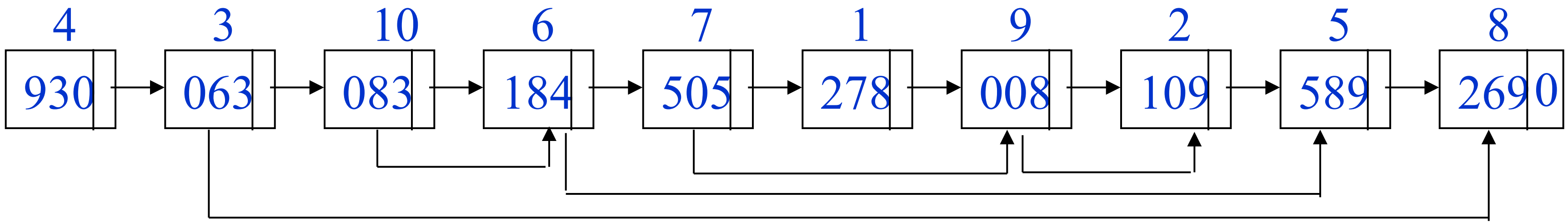
```
template <class T>
int RadixSort (T *a, const int d, const int r, const int n)
{
   int e[r],  f[r];  // queue end and front pointers
   // create initial chain of records starting at first
   int first=1;
   for (int i=1; i<n; i++) link[i]=i+1; // linked into a chain
   link[n]=0;
   for (i=d-1; i>=0; i--)
   { // sort on digit i
      fill(f, f+r, 0); // initialize bins to empty queues
     for (int current=first; current; current=link[current])
        {   // put records into queues
```

```
            int k=digit(a[current], i, r);
            if (f[k]==0) f[k]=current;
            else link[e[k]]=current;
            e[k]=current;
        }
        for (int j=0; !f[j]; j++); // find first nonempty queue
        first=f[j];  int last=e[j];
        for (int k=j+1; k<r; k++) // concatenate remaining queues
            if (f[k] ) {
              link[last]=f[k];  last=e[k];
            }
        link[last]=0;
    }
    return first;
}
```



f[0]= 4	e[0]= 4
f[1]=0	e[1]=0
f[2]=0	e[2]=0
f[3]= 3	e[3]= 10
f[4]= 6	e[4]= 6
f[5]= 7	e[5]= 7
f[6]=0	e[6]=0
f[7]=0	e[7]=0
f[8]= 1	e[8]= 9
f[9]= 2	e[9]= 8





f[0]= 7 e[0]= 2

f[1]=0 e[1]=0

f[2]=0 e[2]=0

f[3]= 4 e[3]= 4

f[4]=0 e[4]=0

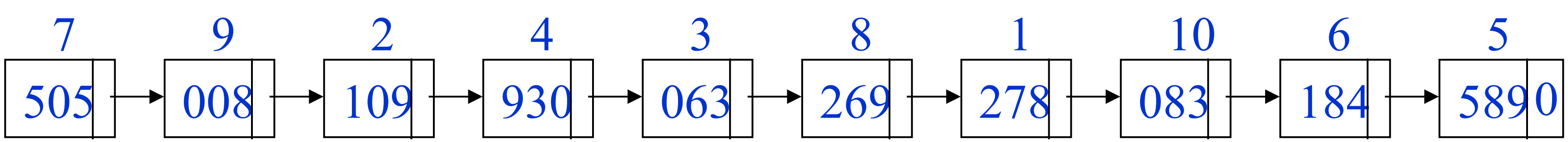
f[5]=0 e[5]=0

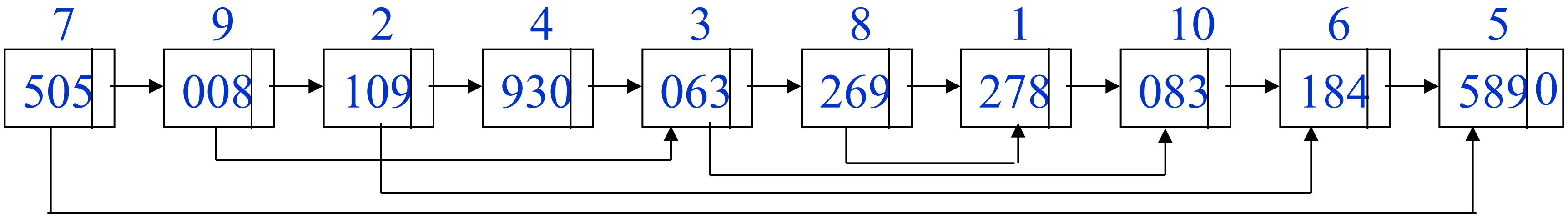
f[6]= 3 e[6]= 8

f[7]= 1 e[7]= 1

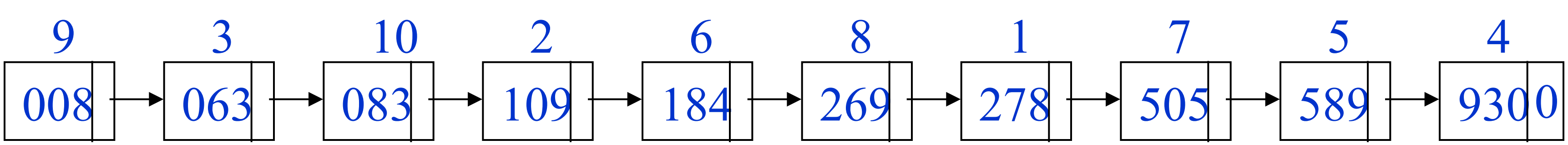
f[8]= 10 e[8]= 5

f[9]=0 e[9]=0





f[0]= 9	e[0]= 10
f[1]= 2	e[1]= 6
f[2]= 8	e[2]= 1
f[3]=0	e[3]=0
f[4]=0	e[4]=0
f[5]= 7	e[5]= 5
f[6]=0	e[6]=0
f[7]=0	e[7]=0
f[8]=0	e[8]=0
f[9]= 4	e[9]= 4



Radix Sort

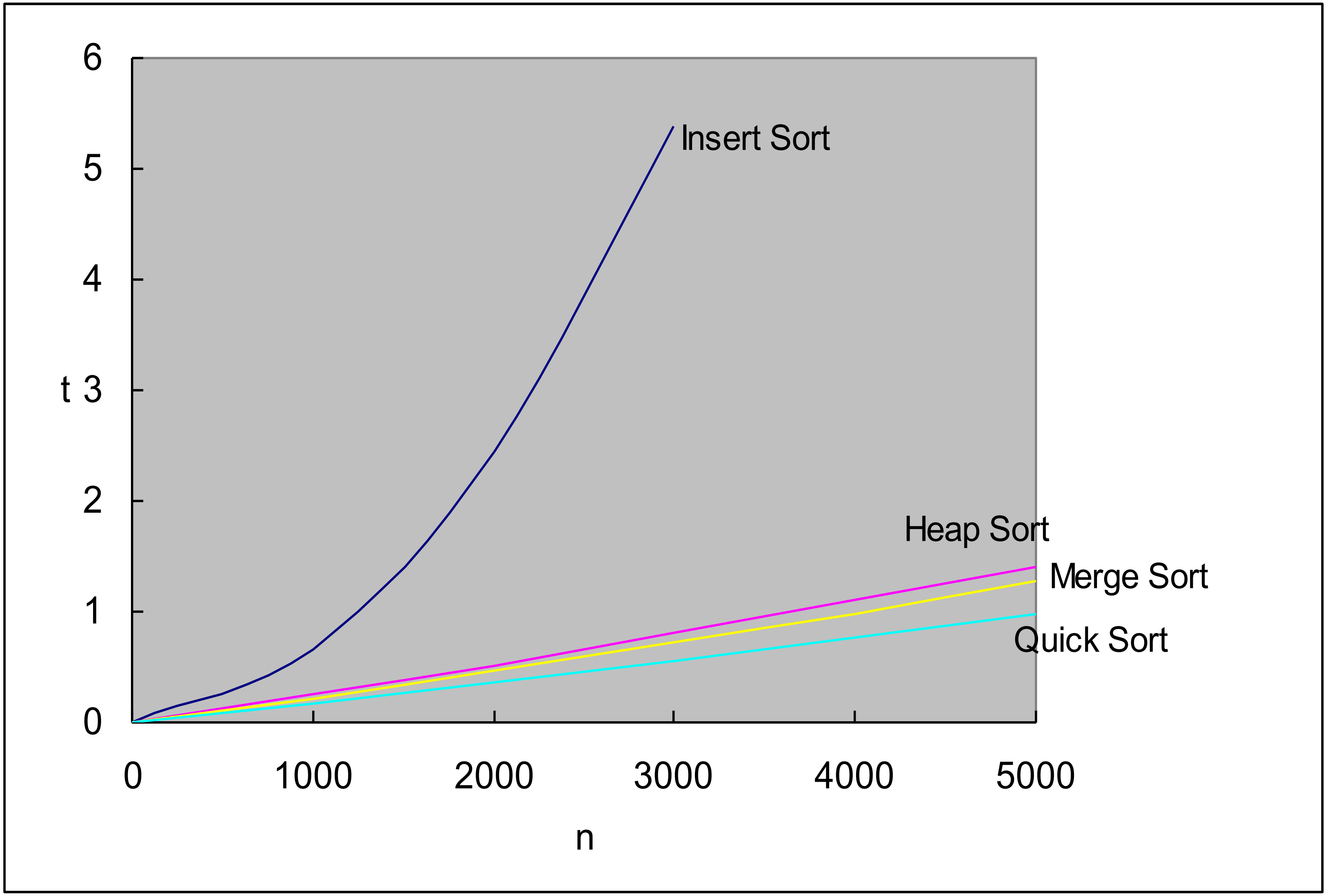
- Increasing the base **r** decreases the number of passes k (e.g. 999)
- Running time
  - k passes over the numbers
  - each pass takes O(N+r)
  - total: O(Nk+rk)
  - r and k are constants: O(N)

Exercises: P422-1, 3, 5

Summary of Internal Sorting

Method	Worst	Average	Working Storage
Insertion Sort	n^2	n^2	O(1)
Heap Sort	n log n	n log n	O(1)
Merge Sort	n log n	n log n	O(n)
Quick Sort	n^2	n log n	O(n) or O(log n)

n	Insert	Heap	Merge	Quick
0	0.000	0.000	0.000	0.000
50	0.004	0.009	0.008	0.006
100	0.011	0.019	0.017	0.013
200	0.033	0.042	0.037	0.029
300	0.067	0.066	0.057	0.045
400	0.117	0.090	0.079	0.061
500	0.179	0.116	0.100	0.079
1000	0.662	0.245	0.213	0.169
2000	2.439	0.519	0.459	0.358
3000	5.390	0.809	0.721	0.560
4000	9.530	1.105	0.972	0.761
5000	15.935	1.410	1.271	0.970



6
5
4
t 3
2
1
0
0
1000
2000
3000
4000
5000
n
Insert Sort
Heap Sort
Merge Sort
Quick Sort

- For average behavior, we can see:
- Quick Sort outperforms the other sort methods for suitably large n.
- the break-even point between Insertion and Quick Sort is near 100, let it be nBreak.
- when n < nBreak, Insert Sort is the best, and when n > nBreak, Quick Sort is the best.
- improve Quick Sort by sorting sublists of less than nBreak records using Insertion Sort.

Experiments: P435-4

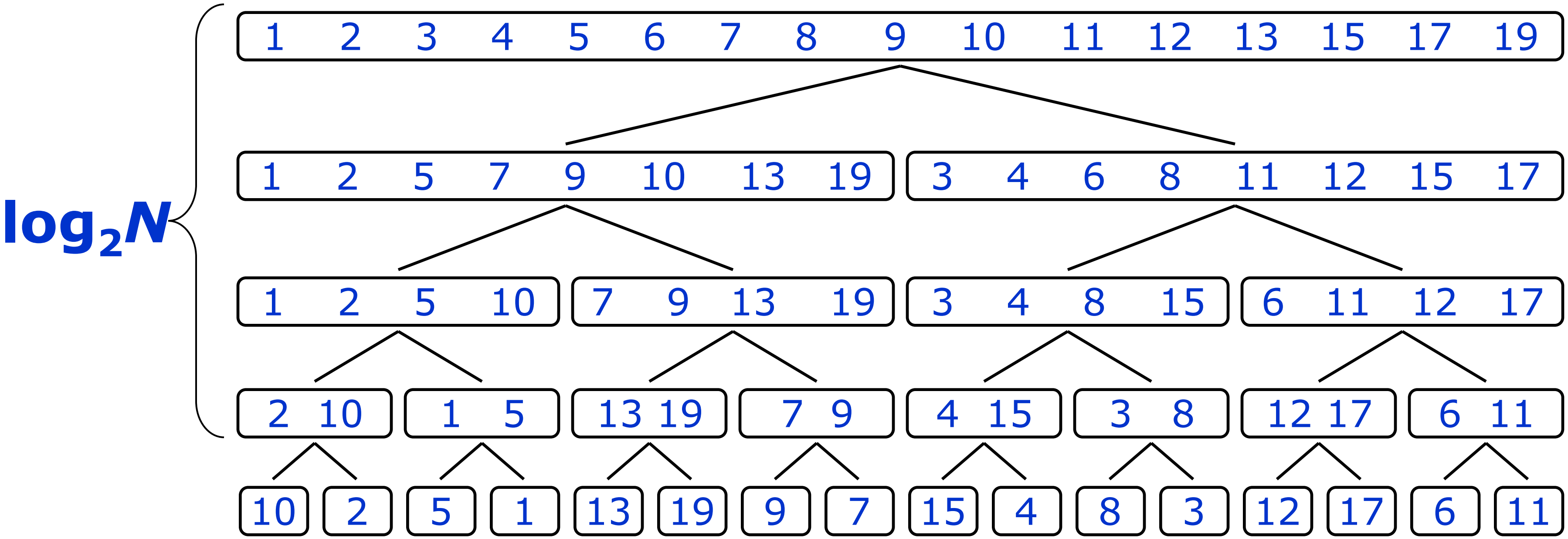
External Sorting

- External-memory algorithms
  - When data do not fit in main-memory
- What does it mean?
  - Sort atomic-operations accomplished with part of data in memory
  - Controllable Disk I/Os
- Which internal sorting algorithm is applicable?
  - Insert sort
  - Exchange sort
  - Select sort
  - Merge sort

External Sorting

- Rough idea:
  - sort pieces that fit in main-memory
    - **known as runs**
  - "merge" them

Merge-Sort Tree



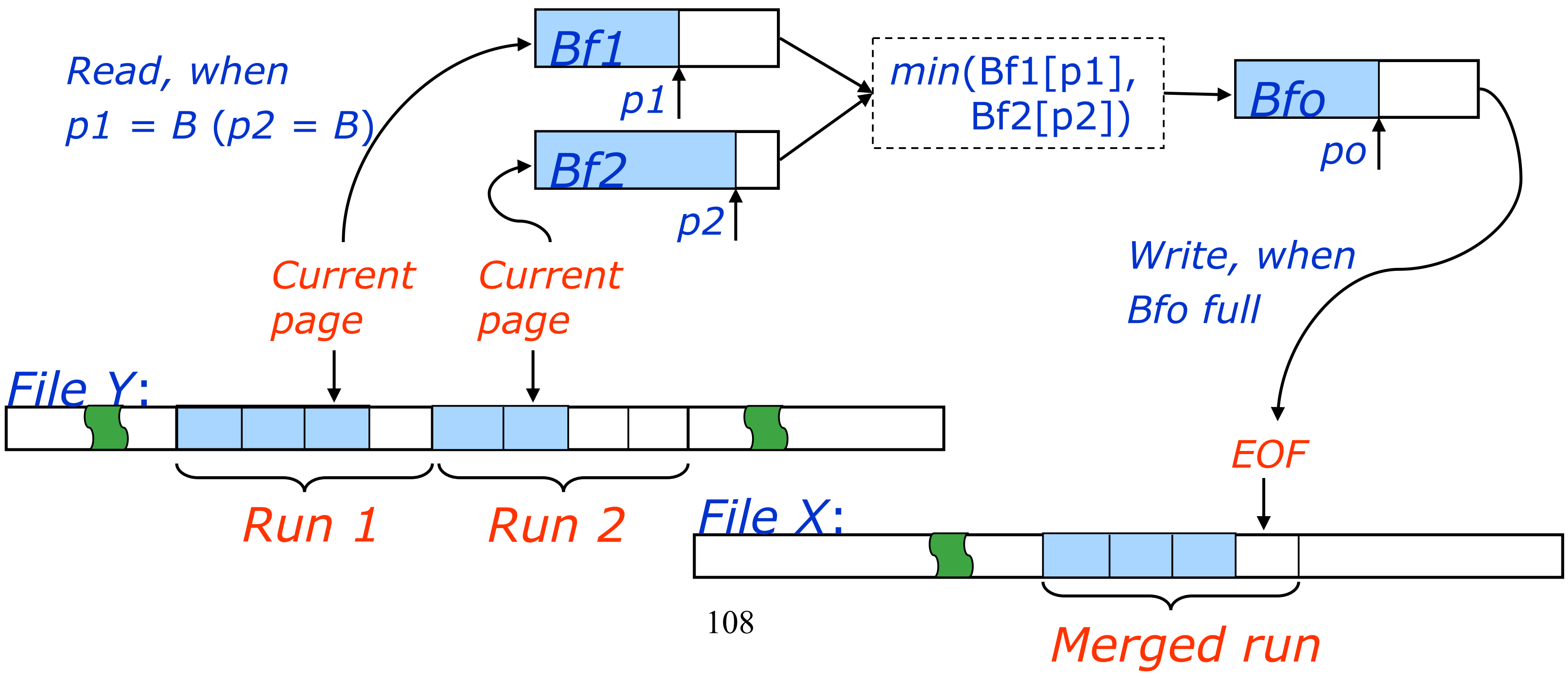
- In each level: merge *runs* (sorted sequences) of size x into runs of size $2x$, decrease the number of runs twofold.
- **What would it mean to run this on a file in external memory?**
  - **One pass, one disk scan!**

External-Memory Merge Sort

- Input file *X*, empty file Y
- *Phase* 1: Repeat until end of file *X*:
  - Read the next *M* elements from *X*
  - Sort them in main-memory
  - Write them at the end of file *Y*
- *Phase* 2: Repeat while there is more than one run in *Y*:
  - Empty *X*
  - *MergeAllRuns*(*Y*, *X*)
  - *X* is now called *Y*, *Y* is now called *X*

External-Memory Merging

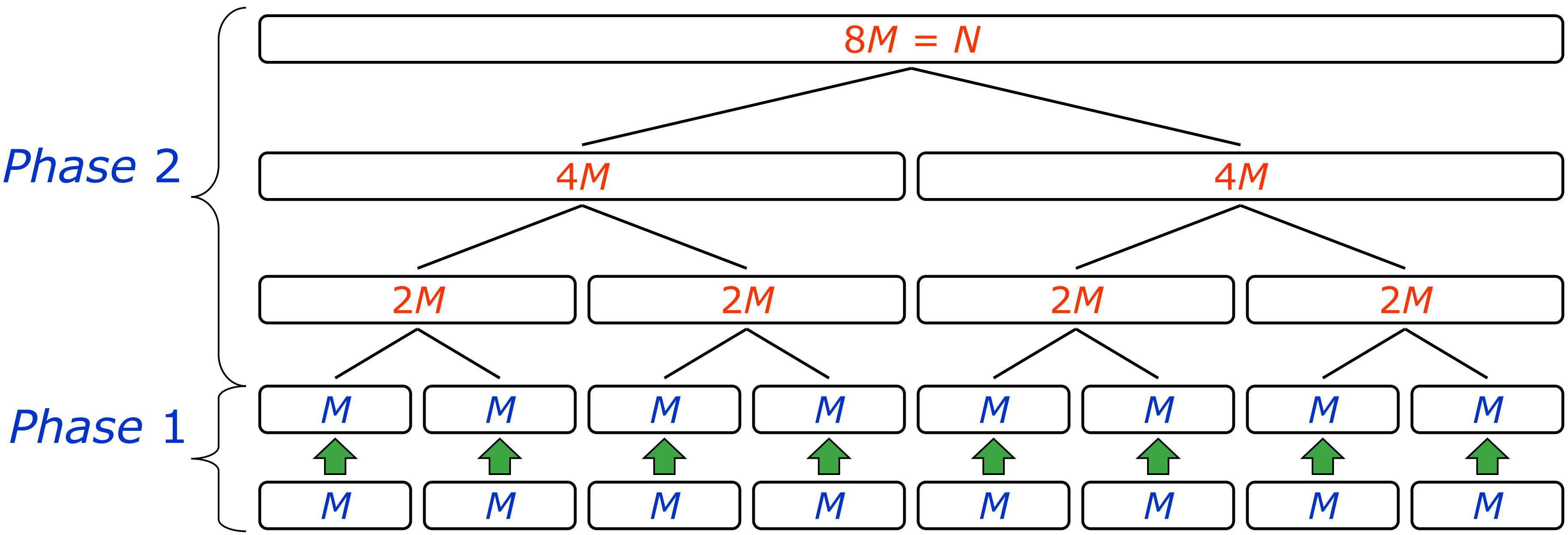
- *MergeAllRuns*(*Y*, *X*): repeat until the end of *Y*:
  - Call *TwowayMerge* to merge the next two runs from *Y* into one run, which is written at the end of *X*
- *TwowayMerge*: uses three main-memory arrays of size *B*



Analysis: Assumptions

- Assumptions and notation:
  - Disk page size:
    - B data elements
  - Data file size:
    - N elements, $n = N/B$ disk pages
  - Available main memory:
    - M elements, $m = M/B$ pages

Analysis



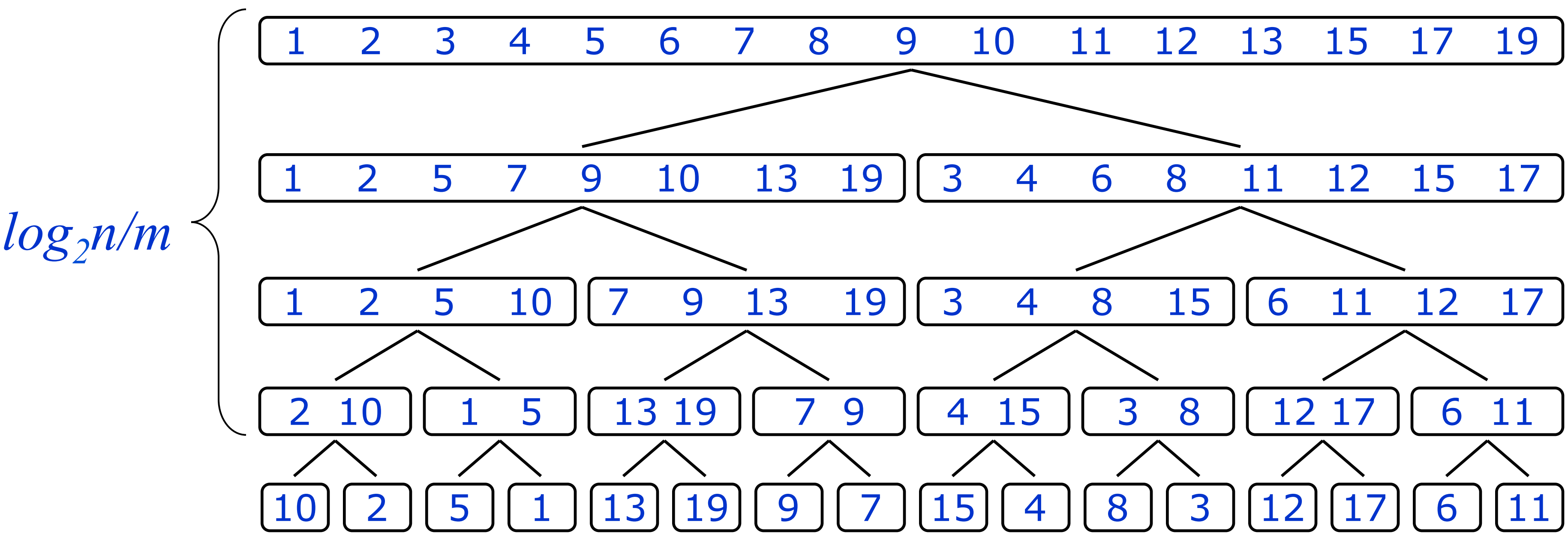
- Phase 1:
  - Read file X, write file Y: $2n = O(n)$ I/Os
- Phase 2:
  - One iteration: Read file Y, write file X: $2n = O(n)$ I/Os
  - Number of iterations: $\log_2 N/M = \log_2 n/m$

Analysis: Conclusions

- Total running time of external-memory merge sort: $O(n \log_2 n/m)$

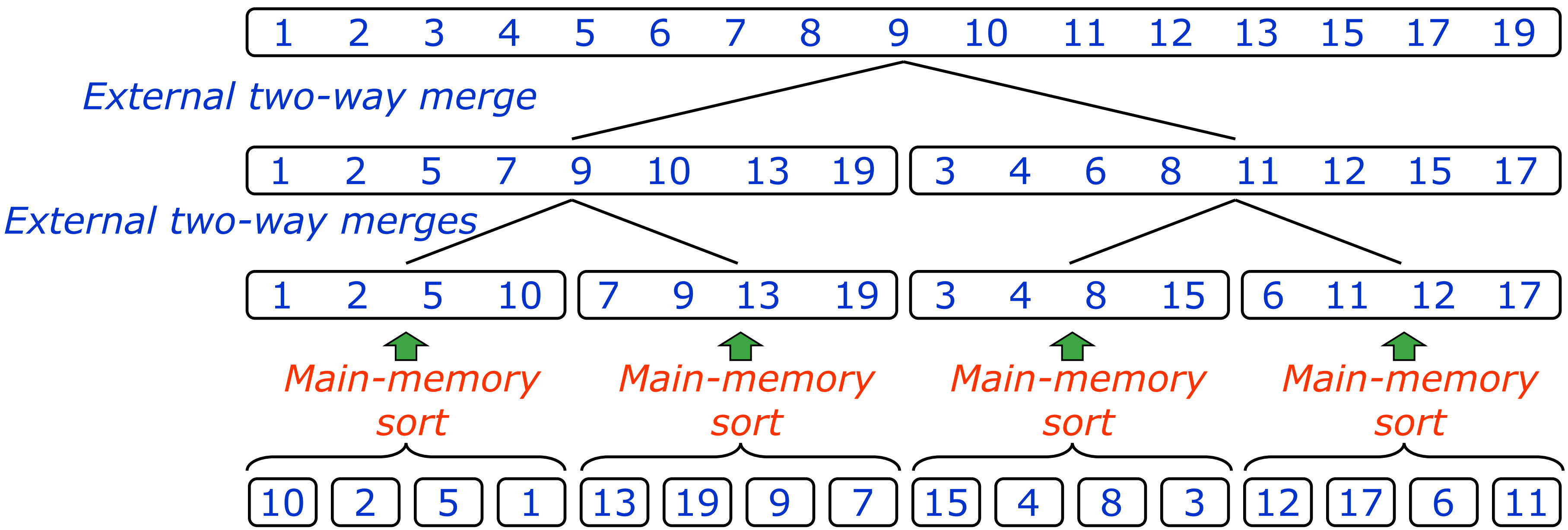
Can we do better?

- $log_2 n/m$ I/Os
  - Decrease n/m?
  - Initial runs – the size of available main memory (M data elements) ????



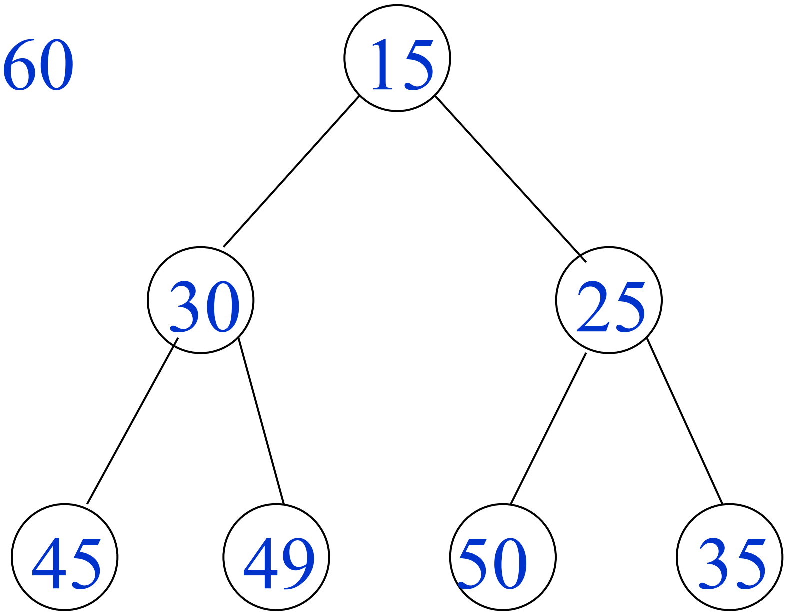
Can we do better?

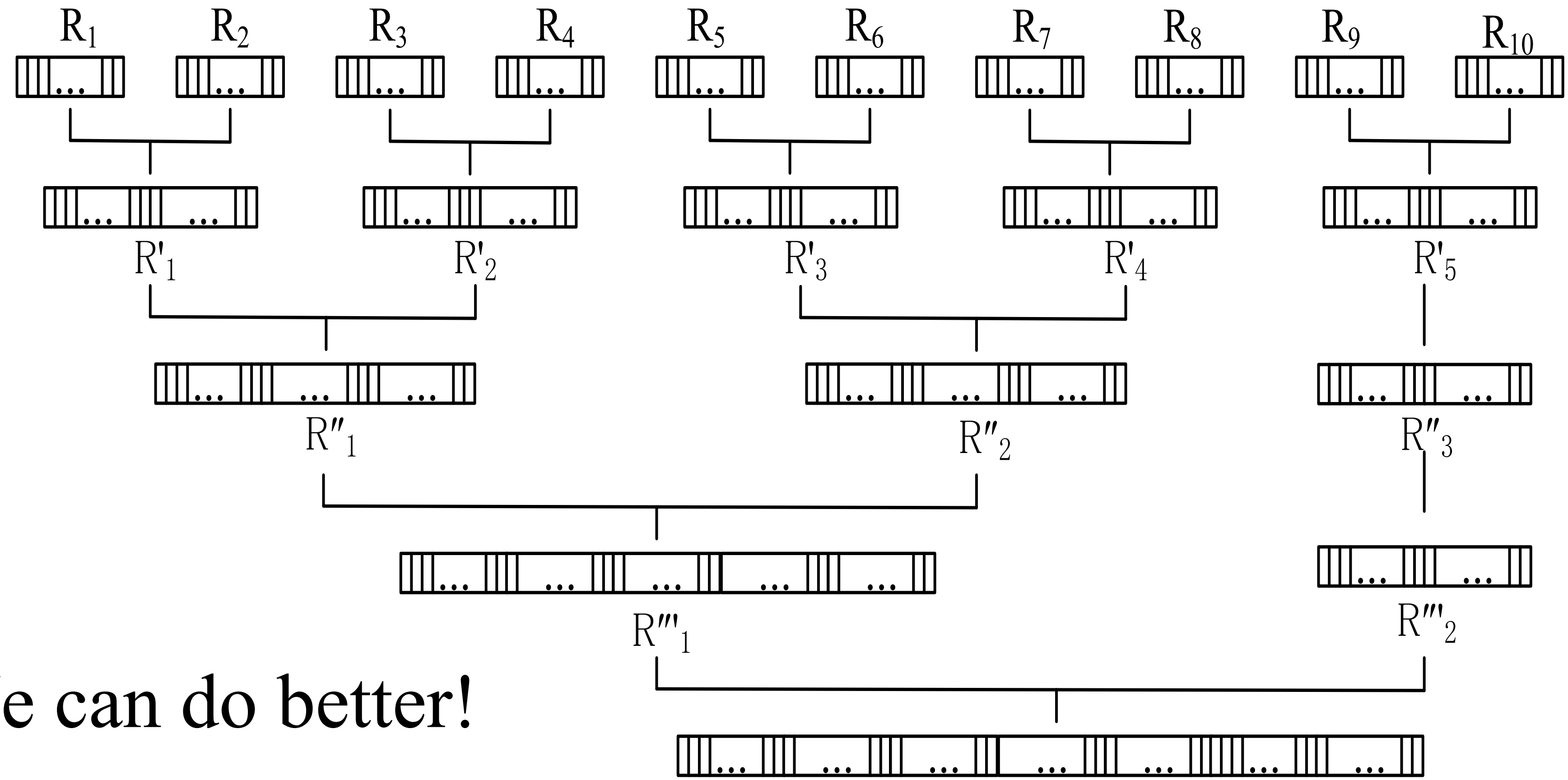
- Idea 1: decrease number of passes
- Increase the size of initial runs!



Run Generation(self study)

27 16 60





We can do better!

- Merge tree : Binary-tree with n leaf elements
- I/O cost =
  - $\Sigma(R_i * L_i)$!
- How to Minimize I/O?

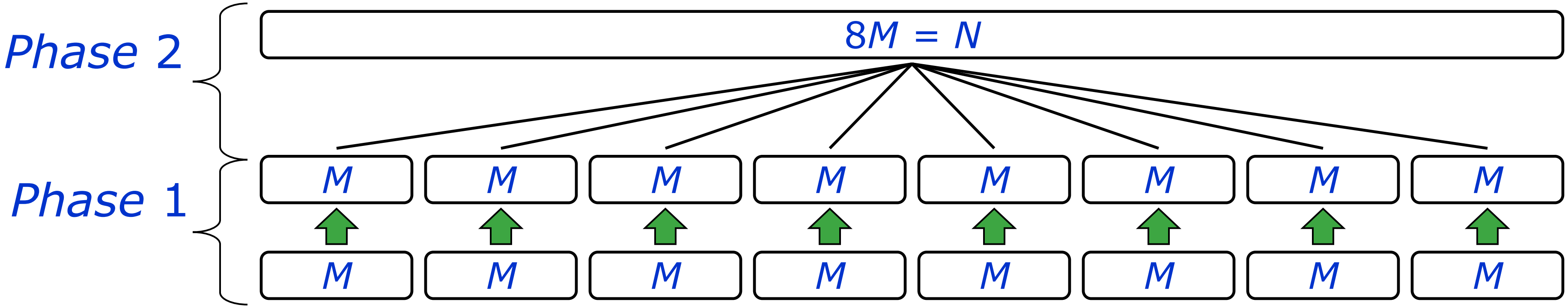
Huffman Tree!

Analysis: Conclusions

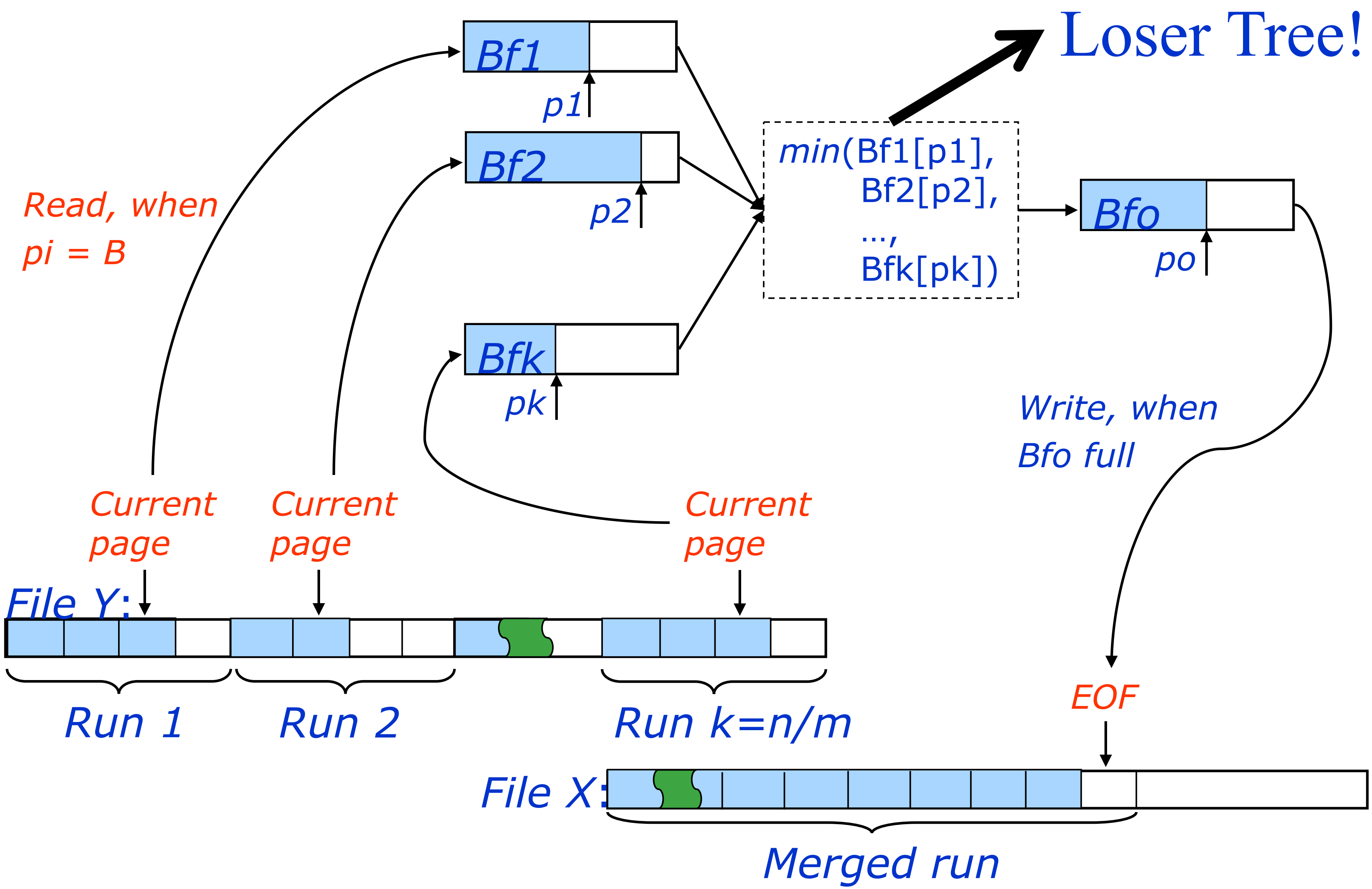
- Total running time of external-memory merge sort: $O(n \log_2 n/m)$
- We can do better!
- Observation:
  - Phase 1 uses all available memory
  - Phase 2 uses just 3 pages out of *m* available!!!

Two-Phase, Multiway Merge Sort

- Idea: merge all runs at once!
  - Phase 1: the same (do internal sorts)
  - Phase 2: perform *MultiwayMerge*(*Y*,*X*)



Multiway Merging



Analysis of TPMMS

- Phase 1: O(n), Phase 2: O(n)

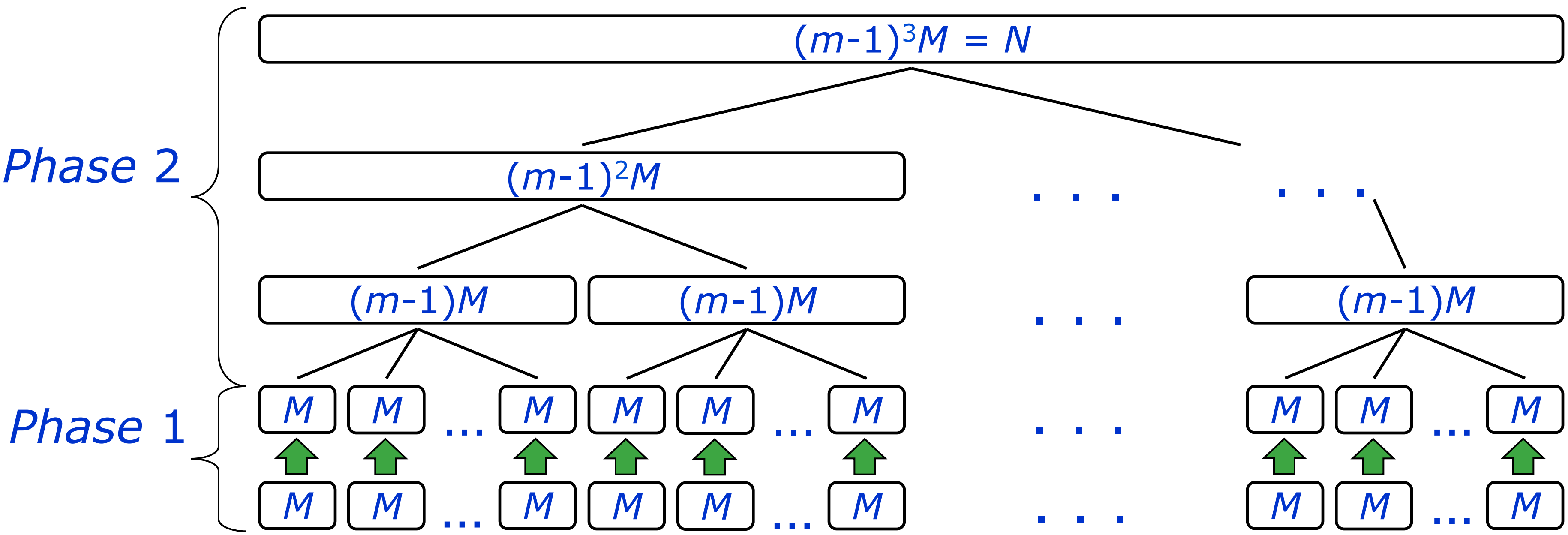
- Total: O(n) I/Os!

- The catch: files only of "limited" size can be sorted
  - Phase 2 can merge a maximum of m-1 runs.
  - Which means: $N/M < m-1$

General Multiway Merge Sort

- What if a file is very large or memory is small?
- General *multiway merge sort*:
  - Phase 1: the same (do internal sorts)
  - Phase 2: do as many iterations of merging as necessary until only one run remains

Analysis



- Phase 1: O(n), each iteration of phase 2: O(n)
- How many iterations are there in phase 2?
  - Number of iterations: $\log_{m-1} N/M = \log_m n$
- Total running time: $O(n \log_m n)$ I/Os

Conclusions

- External sorting can be done in $O(n \log_m n)$ I/O operations for any n
  - This is asymptotically optimal
- In practice, we can usually sort in $O(n)$ I/Os
  - Use two-phase, multiway merge-sort

- **Exercises: P457-2**