Chap. 7 Sorting

7.1 Preliminaries

– input is an array of n elements
– sorting key is integer: sorting in the increasing order of the keys
– internal sorting: all elements are stored in main memory
– external sorting: elements are stored on disk or tape
– comparison-based sorting: comparison ($<$ or $>$) is the only operation applied.

7.2 Insertion Sort

15

3, 6, 9, 12, 21, 22, 67, 78

– $N - 1$ passes are used to insert each element (from the 2^{nd} to the N^{th}) in sequence into correct position
– For pass p, it ensures that the element at positions 0 through p are in sorted order

original	34	8	64	51	32	21	Positions moved
After p=1	8	34	64	51	32	21	1
After p=2	8	34	64	51	32	21	0
After p=3	8	34	51	64	32	21	1
After p=4	8	32	34	51	64	21	3
After p=5	8	21	32	34	51	64	4

– The algorithm:

```
void insertionSort (Array &a) {
  for (p = 1 to N-1) {
        tmp = a[p];
        for (j = p; j > 0 && tmp < a[j-1]; j--)
                a[j] = a[j-1];
        a[j] = tmp;
  }
}
```

– Running time computation
  - Worst case: if the input is in a reversed sorting order:

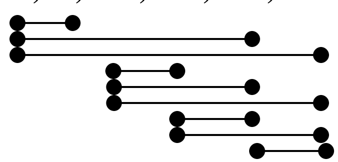
$$\sum_{i=2}^{N} i = \frac{N(N+1)}{2} - 1 = \Theta(N^2)$$

  - Best case: already sorted array: $O(N)$
  - Fast for almost sorted inputs

7.3 A Simple Lower Bound

– *__Inversion__*: an ordered pair (i, j) having the property that $i < j$ but $a[i] > a[j]$

e.g. list: $34, 8, 64, 51, 32, 21$ has 9 inversions



– number of swaps, because swapping two adjacent elements removes one inversion

- If a list has I inversions, the insertion sort requires exactly I time swapping, *i.e.* $O(I+N)$
- *Theorem*: the average number of inversions in an array of N distinct elements is $N(N-1)/4$

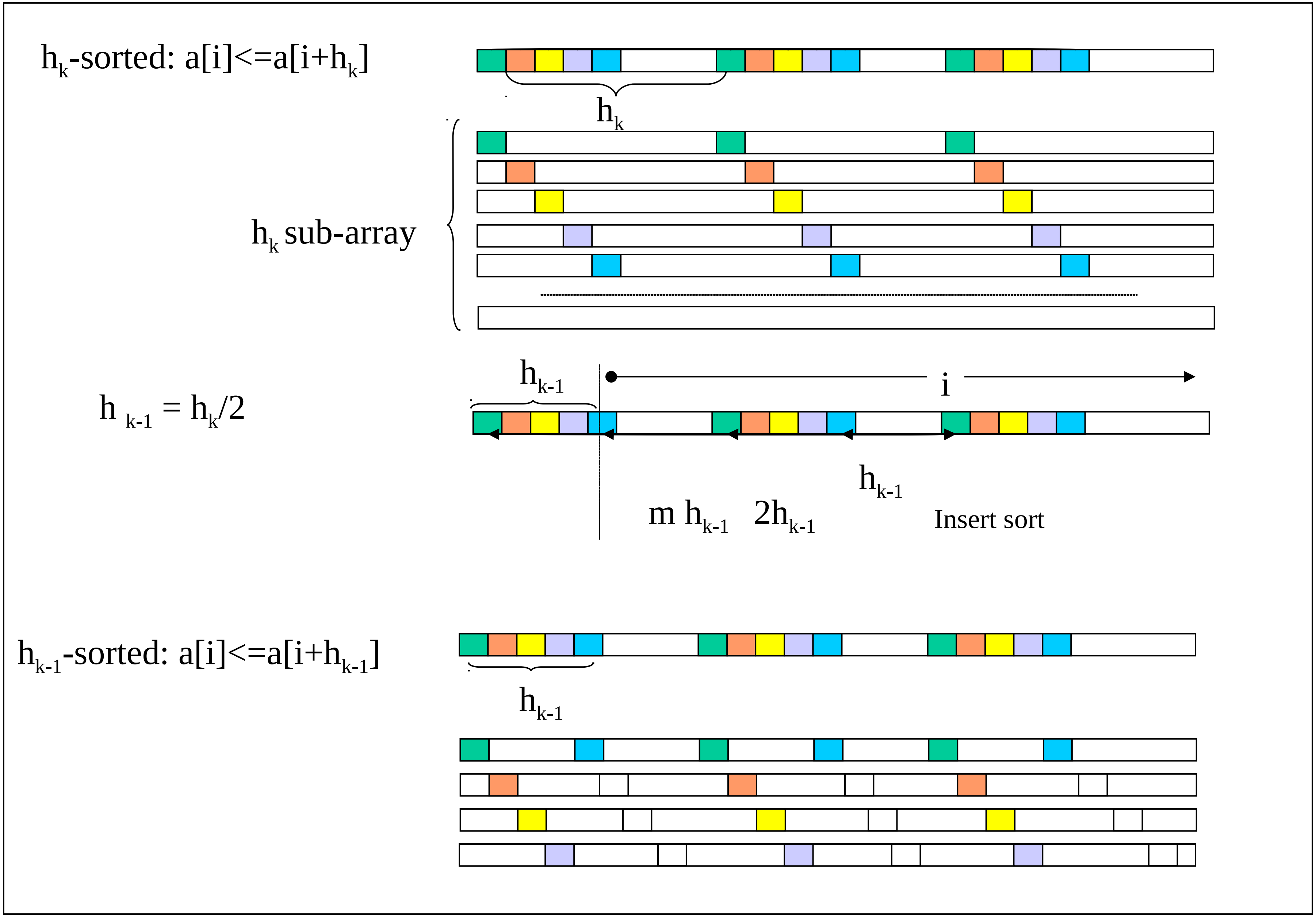
  no. of Combinations $N(N-1)/2$ = *no.inv. + no.inv.Of reversed pairs.*
- average running time: $O(N^2)$
- Theorem: Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average.
- Other algorithms bounded by this lower bound: *Bubble sort, selection sort*.

7.4 ShellSort

- ShellSort is the first algorithm that has an average running time less than $O(N^2)$
- ShellSort uses a pre-defined <u>*increment sequence*</u>, $h_1, h_2, \ldots, h_t$, and sort sub-arrays $a[i + h_k]$ $(k = t,\ t - 1, \ldots, 1)$ sequentially.
- ShellSort property: h_k-*sorted* list is also h_j-*sorted* for $j > k$.
- The performance of the algorithm depends on the increment sequence.
- Example: `[Fig. 7.3]`

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

Figure 7.3 Shellsort after each pass



h_k-sorted: $a[i] <= a[i+h_k]$
h_k
h_k sub-array
h_{k-1}
$h_{k-1} = h_k/2$
i
h_{k-1}
$m\ h_{k-1}$
$2h_{k-1}$
Insert sort
h_{k-1}-sorted: $a[i] <= a[i+h_{k-1}]$
h_{k-1}

- Shell's original sequence: $h_t = N/2, \quad h_k = h_{k+1} / 2$
- The algorithm

```
void shellSort (array  &a) {
    for (h = N/2; h > 0; h /= 2)
      for (i = h; i < N; i++) {
            tmp = a[i];
                    for (j = i; j ≥ h && tmp < a[j - h]; j -= h)
                            a[j] = a[j - h];
                    a[j] = tmp;
            }
      }
```

- Theorem: the worst-case running time of shellsort using shell's increment sequence is $\Theta(N^2)$
- Hibbard increment sequence:

  $1, 3, 7, \ldots, 2^k - 1$
- Theorem: the worst case running time of shellsort using Hibbard increment sequence is $\Theta(N^{3/2})$.

– Sedgewick's increment sequence:

1, 5, 19, 41, 109,…

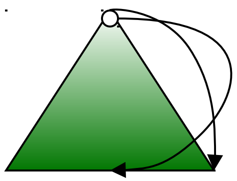
- The terms are either of the form

  $9 \cdot 4^i - 9 \cdot 2^i + 1$ or $4^i - 3 \cdot 2^i + 1$
- Worst-case running time $O(N^{5/4})$
- Average case running time $O(N^{7/6})$

Start	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 8-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 4-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 2-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 1-sort	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 7.5 Bad case for Shellsort with Shell's increments (positions are numbered 1 to 16)

7.5 HeapSort



i

- Using priority queue to sort in $O(N \log N)$ time:
  - 1.) Build a priority queue from the input array: $O(N)$
  - 2.) deleteMin N times, generating a sequence in sorted order.
- To avoid using an extra array, the return value of deleteMin can be put back into the last place of the heap (emptied by deleteMin).

  If the heap is sorted in decreasing order (the root has the maximum element), the result will be in the same array in an increasing order. [Fig. 7.6 - 7.7]
- C++ implementation: [Fig. 7.8]

```
template <class Comparable>
     void heapsort( vector<Comparable> & a )
     {
/* 1*/     for( int i = a.size( ) / 2; i >= 0; i-- )  /* buildHeap */
/* 2*/         percDown( a, i, a.size( ) );
/* 3*/     for( int j = a.size( ) - 1; j > 0; j-- )
           {
/* 4*/         swap( a[ 0 ], a[ j ] );                   /* deleteMax */
/* 5*/         percDown( a, 0, j );
           }
     }
```

- The worst-case running time

$$2\ N \log N - O(N)$$

- Theorem: the average number of comparisons used to heapsort a random permutation of N distinct items is

$$2\ N \log N - O(N \log \log N)$$

7.6 MergeSort

– Merging two sorted lists:

- Use a moving pointer for each list, at each step, copy the smaller element of the two pointed by the current pointers to a new array, and advance the pointer of the new array.
- Example: `[pp. 264 - 265]`

1	13	24	26		**2**	15	27	38									
1	**13**	24	26		**2**	15	27	38		1							
1	**13**	24	26		2	**15**	27	38		1	2						
1	13	**24**	26		2	**15**	27	38		1	2	13					
1	13	**24**	26		2	15	**27**	38		1	2	13	15				
1	13	24	**26**		2	15	**27**	38		1	2	13	15	24			
1	13	24	26		2	15	**27**	38		1	2	13	15	24	26		
1	13	24	26		2	15	27	38		1	2	13	15	24	26	27	38

- $O(N)$ algorithm.

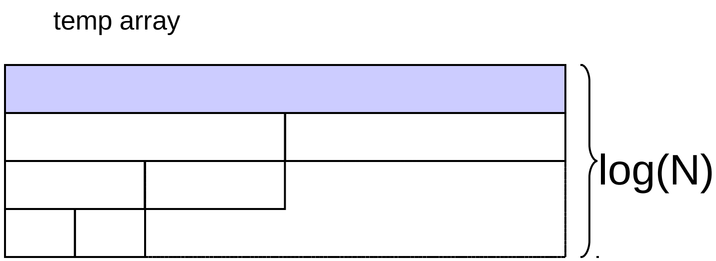
– MergeSort

- If $N = 1$, done;

  otherwise, recursively mergeSort the first half and the second half;

  Then merge these two (sorted) halves.

- A divide-and-conquer algorithm



- C++ implementation: [Fig. 7.9 - 7.10]

```
/**Mergesort algorithm (driver).*/
        template <class Comparable>
        void mergeSort( vector<Comparable> & a )
        {
            vector<Comparable> tmpArray( a.size( ) );
            mergeSort( a, tmpArray, 0, a.size( ) - 1 );
        }
```

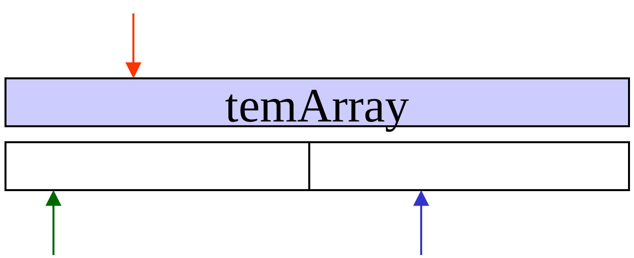
```
/* Internal method that makes recursive calls. a is an
array of Comparable items. tmpArray is an array to
place the merged result. left is the left-most index of
the subarray. right is the right-most index of the
subarray.*/

    template <class Comparable>

    void mergeSort( vector<Comparable> & a,

          vector<Comparable> & tmpArray, int left,
int right )

    {
      if( left < right )
      {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
      }
    }
```




```
/* Internal method that merges two sorted halves of a subarray. a is an
array of Comparable items. tmpArray is an array to place the merged
result. leftPos is the left-most index of the subarray. rightPos is the
index of the start of the second half. rightEnd is the right-most index of
the subarray. */

    template <class Comparable>

    void merge( vector<Comparable> & a, vector<Comparable>
& tmpArray, int leftPos, int rightPos, int rightEnd )

    {
      int leftEnd = rightPos - 1;
      int tmpPos = leftPos;
      int numElements = rightEnd - leftPos + 1;
      // Main loop
      while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[ leftPos ] <= a[ rightPos ] )
          tmpArray[ tmpPos++ ] = a[ leftPos++ ];
        else
          tmpArray[ tmpPos++ ] = a[ rightPos++ ];

      while( leftPos <= leftEnd )    // Copy rest of first half
        tmpArray[ tmpPos++ ] = a[ leftPos++ ];

      while( rightPos <= rightEnd )  // Copy rest of right half
        tmpArray[ tmpPos++ ] = a[ rightPos++ ];
      // Copy tmpArray back
      for( int i = 0; i < numElements; i++, rightEnd-- )
        a[ rightEnd ] = tmpArray[ rightEnd ];
    }
```

– Analysis

- The direct analysis

  for simplicity of analysis, let $N = 2^k$ (i.e. $k = logN$), and the running time function of mergeSort be $T(N)$.

$$\Rightarrow \begin{cases} T(1) = 1 \\ T(N) = 2\,T(N/2) + N \end{cases}$$

or $T(N) = 2\,T(N/2) + N = 4\,T(N/4) + 2\,N = \ldots = 2^k\,T(N/2^k) + k\,N$

$= N\,T(1) + k\,N = N + N\,logN = O(N\,logN)$

- An alternative approach

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

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1, \frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + 1,$$

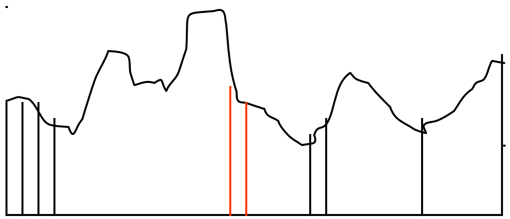
$$\ldots \frac{T(2)}{2} = \frac{T(1)}{1} + 1 \qquad \Rightarrow \frac{T(N)}{N} = T(1) + k \quad \Rightarrow T(N) = O(N \log N)$$

  Add all equations

- The analysis for N not a power of 2 is similar
- In practice, mergeSort is not very fast, due to the extra computation in merging and array copying.

7.7 Quicksort

– A divide-and-conquer algorithm
  - worst-case running time: $O(N^2)$
  - average-case running time: $O(N \log N)$
  - In practice, quicksort is the fastest sorting algorithm for large input arrays

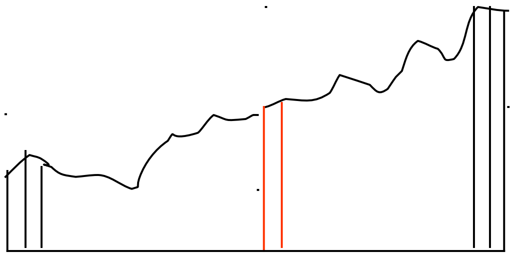


– Basic steps

For an input array of N element: S

1.) if N is 0 or 1, return

2.) pick an element v in S (v is called pivot)



3.) Let $S - \{v\} = S_1 \cup S_2$

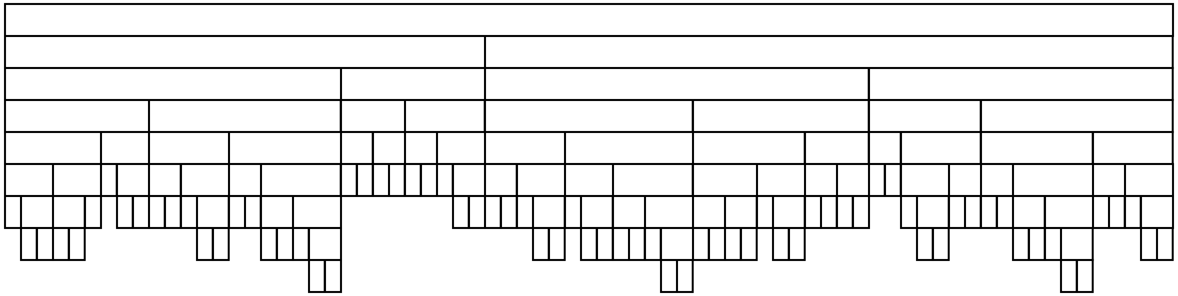
$S_1 = \{x \in S - \{v\}: x \leq v\}$

$S_2 = \{x \in S - \{v\}: x \geq v\}$

4.) return $\{quicksort\ (S_1)\ v\ quicksort\ (S_2)\}$

– Example: `[ Fig. 7.11]`

				pivot					
13	81	92	43	65	31	57	26	75	0
13	43	57	31	26	0	65	75	81	92
0	13	26	31	43	57	65	75	81	92



- Picking the pivot
  - pick the *1st* element: can lend to $O(N^2)$ worst case for pre-sorted arrays
  - pick at a random position: a generally good and safe pick, but need to use a random generator.
  - Pick the median of the three elements at 0, $N - 1$ and $\lceil N/2 \rceil$ positions: a good choice in general.

- Partitioning strategy

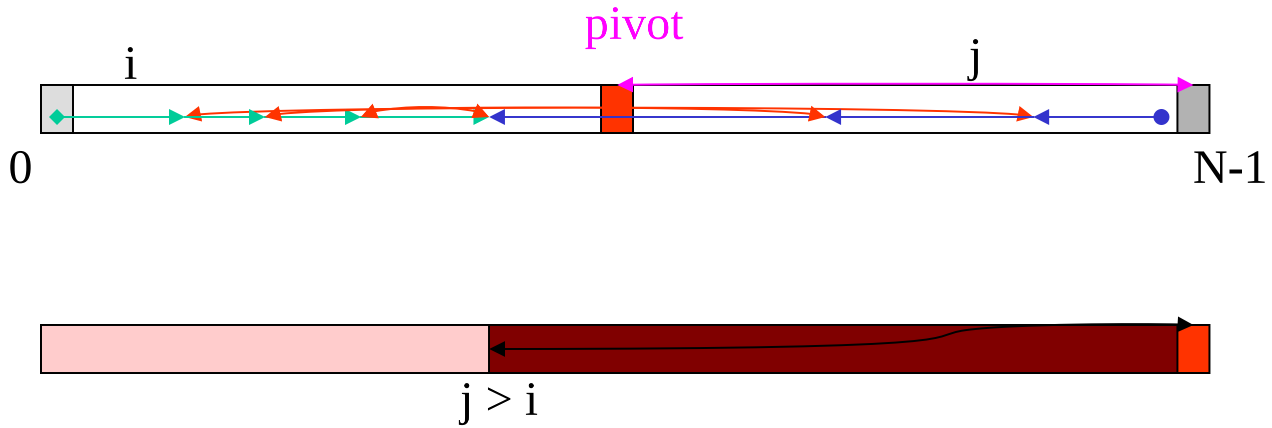
  1.) swap the pivot v and the last element $a[N - 1]$

  2.) $i = 0$; $j = N - 2$

  3.) while ($i < j$)

  - move i right, skipping over elements that are smaller than v.
  - move j left, skipping over elements that are greater than v.
  - when i & j stopped, swap the elements at i & j.

  4.) swap $a[i]$ and $a\ [N - 1]$



- Example: `[pp. 272 - 273]`

8 ← i	1	4	9	0	3	5	2	7 ← j	6
8 ← i	1	4	9	0	3	5	2 ← j	7	6
2 ← i	1	4	9	0	3	5	8 ← j	7	6
2	1	4	9 ← i	0	3	5 ← j	8	7	6
2	1	4	5 ← i	0	3	9 ← j	8	7	6
2	1	4	5	0	3 ← j	9 ← i	8	7	6
2	1	4	5	0	3	6	8	7	9

- Handling equal elements: both i & j move (skip) over elements that are equal to the pivot (for balanced partitioning)

– Small arrays
  - When N is small, quicksort is slower than insertion sort.
  - Solution: in the recursion process, when N is smaller than a given number (say 10 or 20), switch to insertion sort.

- C++ implementation:
  - [Fig. 7.13 - 7.15]

```
/**Quicksort algorithm (driver). */
template <class Comparable>
void quicksort( vector<Comparable> & a )
{
    quicksort( a, 0, a.size( ) - 1 );
}
/**Standard swap*/
template <class Comparable>
inline void swap( Comparable & obj1, Comparable & obj2 )
{
    Comparable tmp = obj1;
    obj1 = obj2;
    obj2 = tmp;
}
/** Return median of left, center, and right.Order these and hide the pivot. */
template <class Comparable>
const Comparable & median3( vector<Comparable> & a, int left, int right )
{
    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 ] );
    return a[ right - 1 ];
}
```

```
/*  Internal quicksort method that makes recursive calls.
* Uses median-of-three partitioning and a cutoff of 10.
* a is an array of Comparable items.
* left is the left-most index of the subarray.
* right is the right-most index of the subarray.       */
    template <class Comparable>
    void quicksort( vector<Comparable> & a, int left, int right )
    {
/* 1*/    if( left + 10 <= right )
          {
/* 2*/        Comparable pivot = median3( a, left, right );
              // Begin partitioning
/* 3*/        int i = left, j = right - 1;
/* 4*/        for( ; ; ) {
/* 5*/            while( a[ ++i ] < pivot ) { }
/* 6*/            while( pivot < a[ --j ] ) { }
/* 7*/            if( i < j )
/* 8*/                swap( a[ i ], a[ j ] );
/* 9*/            else  break;
              }
/*10*/        swap( a[ i ], a[ right - 1 ] );  // Restore pivot
/*11*/        quicksort( a, left, i - 1 );     // Sort small elements
/*12*/        quicksort( a, i + 1, right );    // Sort large elements
          }
          else  // Do an insertion sort on the subarray
/*13*/        insertionSort( a, left, right );
    }
```

– Analysis

- $T(N) = T(i) + T(N - i - 1) + c\,N$

  where i is the number of elements in S_i, c is a constant.
- Worst case: i is always 0.

  $T(N) = T(N - 1) + c\,N = T(N - 2) + c(N - 1) + c\,N = \ldots = T(1) + c\sum_{2}^{N} k = O(N^2)$
- Best case: i is always half of the array size (*i.e.* v is always the median element): $T(N) = 2\,T(N/2) + c\,N = \ldots = N\,T(1) + c\,N\,logN = O(N\,logN)$
- Average case:

  on average, $T(i) = T(N - i - 1) = \frac{1}{N}\sum_{j=0}^{N-1} T(j)$

  $$T(N) = \frac{2}{N}\sum_{j=0}^{N-1} T(j) + cN \quad \text{or} \quad NT(N) = 2\sum_{j=0}^{N-1} T(j) + cN^2$$

  and

  $$(N-1)T(N-1) = 2\sum_{j=0}^{N-2} T(j) + c(N-1)^2$$

  Subtract: $T(N) = \frac{N+1}{N}T(N-1) + 2c = \ldots$

  $$= \frac{N+1}{2}T(1) + 2c(N+1)\sum_{i=3}^{N+1}\frac{1}{i}$$

  $$= O(N \log N)$$

– Quick selection problem

- Selecting the k^{th} smallest element in an array S of size N

  1.) If $(N = 1)$ return

  2.) pick a pivot $v \in S$

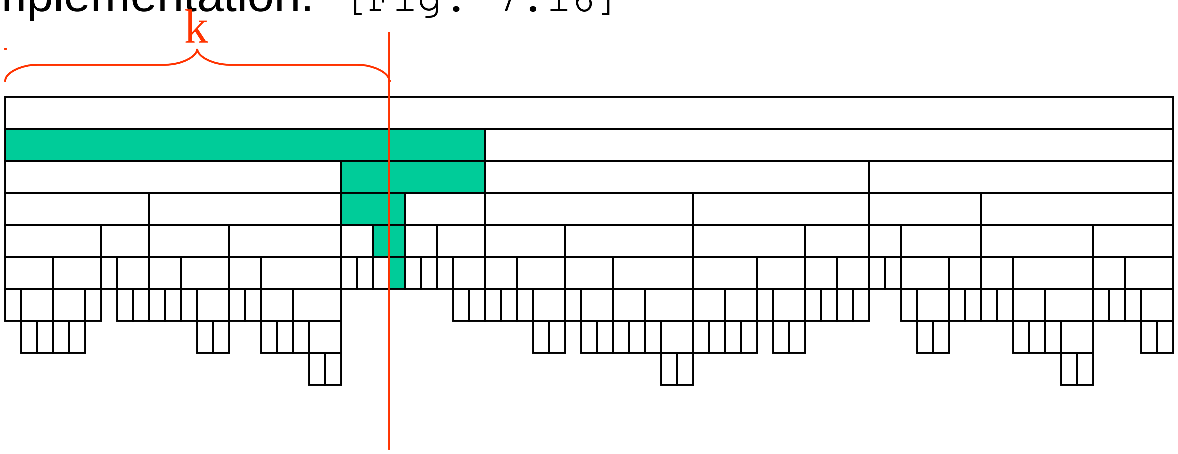
  3.) partition S into S_1 & S_2

  4.) If $k \leq |S_1|$, return quickSelect (S_1, k)

  If $k = 1 + |S_1|$, return v

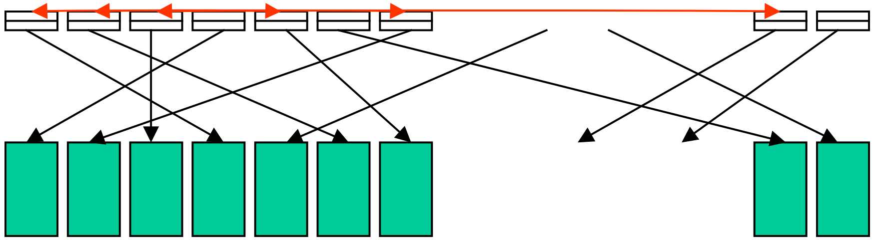
  otherwise quickSelect $(S_2, k - |S_1| - 1)$

- implementation: `[Fig. 7.16]`



– Indirect Sorting

- When data records are large, moving or swapping is costly.
- Use a separate pointer array (may also carry the keys). Sorting is done in the pointer array only.
- Data array can be rearranged after the pointer array is sorted (may need an extra array): $O(N)$ data movements.



7.9 A General Lower-Bound for Sorting

- Decision tree
  - A decision tree is a binary tree.

    Each node represents a set of possible orderings, and each edge represents one comparison result.
  - The root represents the initial state, *i.e.* all possible orderings (permutations) of N elements.
  - Each leaf node represents one ordering as the result of a sequence of comparisons represented by the path from the root to the leaf node.

$3*2*1 = 6$

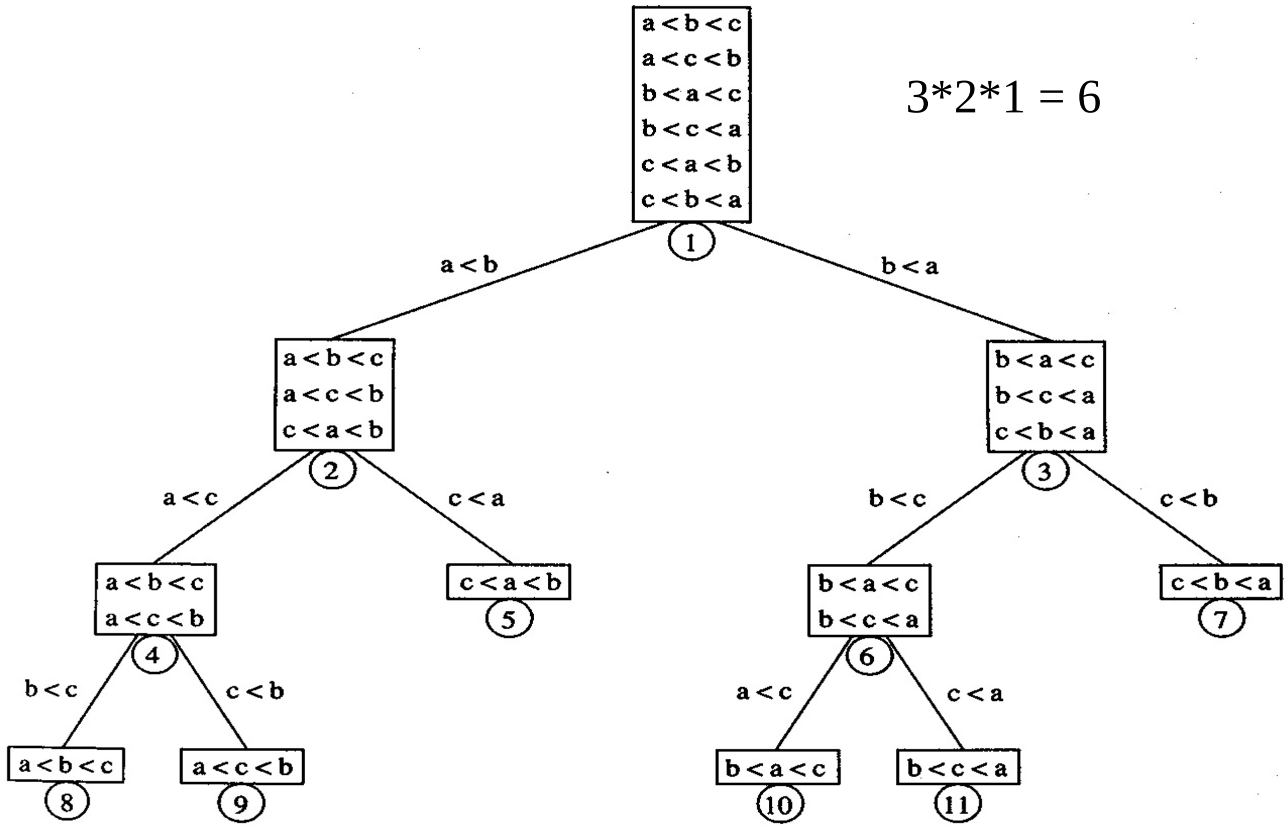
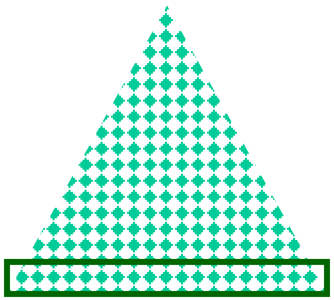


Figure 7.20 A decision tree for three-element insertion sort

– Lemma: Let T be a binary tree of height h. Then T has at most 2^h leaves. (*Proof:* by induction.)

– Lemma: A binary tree with L leaves must have height at least $\lceil log L \rceil$

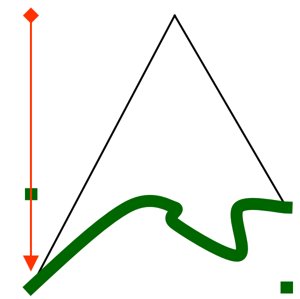


L (no. of leaves) ~ h (height) relation

$L < 2^h \quad \leftarrow h$

$L \rightarrow \quad \lceil log L \rceil < h$

Worst case = deepest path



– Theorem: Any comparison-based sorting algorithm requires at least $\lceil log N! \rceil$ comparisons in the worst case.

Proof: There are total $N!$ orderings, $i.e.$ $N!$ leaves.

– Theorem: Any comparison-based sorting algorithm requires $\Omega(N log N)$ comparisons.

Proof:

$$\log N! = \log N + \log N - 1 + \ldots + \log 2 + \log 1$$

$$\geq \frac{N}{2} \log N/2 = \Omega(N \log N)$$

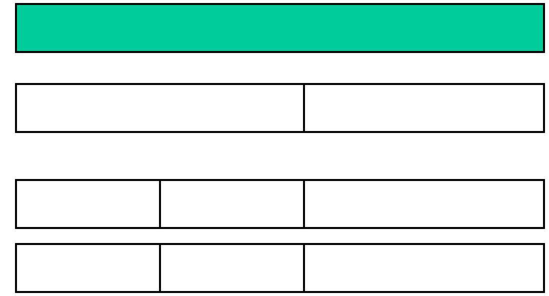
7.10 Bucket Sort

- A linear algorithm when the input $\{A_i\}$ is all positive integers bounded by $M > 0$.
- The algorithm:

```
int count[M]
initialize all elements of count to 0
for (i = 1 to N)
  count[Ai]++;
print array count;
```

- Bucket Sort is not bounded by the general sorting lower bound because it is not a comparison-based algorithm, and its input is a restricted type.

7.11 External Sorting



- Internal Sorting: random access
- External storage: sequential access
  - Not directly addressable (*e.g.* tape)
  - Slow
  - External sorting is device dependent.
- External sorting with tapes (A simple merge sort)
  1) Four tapes are used: T_{a1}, T_{a2}, T_{b1}, $T_{b2,}$
     Initial input is on T_{a1}
  2) The internal memory can hold and sort M elements at a time. The result of each internal sorting (of M elements) is called a run. Each run is stored on tape T_{b1} or T_{b2} in an alternate fashion.
  3) Take a run from both T_{b1} & T_{b2} at a time, merge them and write the result (a run twice as long) to T_{a1} or T_{a2} alternately.
  4) Continue this merging process from T_{a1} & T_{a2} to T_{b1} & T_{b2}, until all are sorted.
  - Example: `[pp. 290 - 291]`