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

CSCI2100B CSCI2100S DATA STRUCTURES

Spring 2011

Sorting Algorithms

Last updated: **09/02/2011**

Tang Wai Chung, Matthew

Contents

- Motivations & terminologies
- **Elementary** sorting algorithms
  - **Bubble** sort
  - **Insertion** sort
  - **Selection** sort
  - **Shell** sort *
- **Merge** sort
- **Quicksort**
- **Lower bound** of runtime complexity for general sorting algorithm *

Important examples are marked with

Motivation

- How items are stored in computer memory often has **profound** influence on speed and simplicity of algorithms.
- **Imagine how hard it would be to use a dictionary that is not alphabetized!**
- Sorting is clearly a **practical** problem.
  - But it also makes a **valuable** case study of how to attach programming problems in general.
- Sorting techniques provide **excellent** illustrations of the general ideas in the analysis of algorithms.

Terminologies

- We are going to study methods of sorting **files** of **items** containing **keys**.
- Each **item**(record) contains a **key**, which is a small part of the item, and are used to control the sort.
  - The remainder of the item is called satellite data.
- **Sorting**: rearrange the items such that their keys are ordered.
  - e.g. numerical order, alphabetical order, lexicographical order
- **Internal sort**: sorting takes place entirely within the main memory of the computer.
- **External sort**: handle massive amounts of data that cannot fit into the main memory. It requires a slower kind of memory (like hard-disk) for the sorting.

Stability

A sorting method is said to be **stable** if it preserves the relative order of items with duplicated keys in the file.

Adams	1
Black	2
Brown	4
Jackson	2
Jones	4
Smith	1
Thompson	4
Washington	2
White	3
Wilson	3

Adams	1
Smith	1
Washington	2
Jackson	2
Black	2
White	3
Wilson	3
Thompson	4
Brown	4
Jones	4

unstable

Adams	1
Smith	1
Black	2
Jackson	2
Washington	2
White	3
Wilson	3
Brown	4
Jones	4
Thompson	4

stable

Note: only some of the sorting algorithms in our discussion are stable.

Permutation and *k*-Permutation

- A **permutation** of a finite set *S* is an **ordered** sequence of all the elements of *S*, with each element appearing exactly once.
- For example, if *S* = {*a*, *b*, *c*}, there are 6 permutations of *S*:

  abc, acb, bac, bca, cab, cba

- There are *N*! permutations of a set of *N* elements.
- A ***K*-permutation** of *S* is an ordered sequence of *K* elements of *S*, with no element appearing more than once in the sequence.
  - An ordinary permutation is just an *N*-permutation of an *N*-set.)
- The twelve 2-permutations of the set *a*, *b*, *c*, *d* are

  ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc

Inversion

- An inversion in an array of numbers is any ordered pair (i, j) having the property that $i < j$ but $a[i] > a[j]$.
- For example, the input list 34, 8, 64, 51, 32, 21 has nine inversions, namely (34,8), (34,32), (34,21), (64,51), (64,32), (64,21), (51,32), (51,21) and (32,21).
- **Intuition**: We need to **swap** the element $a[i]$ and $a[j]$ so that the array is closer to be sorted. Thus more inversions imply more swaps are needed.
- Notice that this is exactly the number of swaps that needed to be (implicitly) performed by insertion sort.

Assumptions

- We assume the following in our discussion of the sorting algorithms:
  - **Internal** - Each algorithm will be passed an array containing the elements and an integer containing the number of elements stored **entirely** in the main memory.
  - **Validity** -- We will assume that *n*, the number of elements passed to our sorting routines, has already been checked and is legal.
  - **Ordering** -- We require the existence of the < and > operators, which can be used to place a consistent ordering on the input.

Elementary Sorting Algorithms

- **Easy** to implement.
- Take $O(N^2)$ time to sort N items in **worse** cases.
- Yet suitable for:
  - occasions which the sort problem has to be solved only **once**.
  - **small** files where the number of items to be sorted is not too large. (< 500)
    Reasons: sophisticated algorithms generally incur **overheads** that makes them slower in small files.

Selection Sort

- It is also called **straight selection** or **push-down** sort.
- The algorithm works as follows:
  1. Find the **minimum** value in the list (select)
  2. **Swap**(exchange) it with the value in the **first** position
  3. **Repeat** the steps above for the remainder of the list (to 2^{nd}, 3^{rd}, …., $(n - 1)^{th}$ positions) until the entire array is sorted.

Selection Sort: Example

	34	8	64	51	32	21	#ex	#cm
1	34	(8)	64	51	32	21	1	5
2	8	34	64	51	32	(21)	1	4
3	8	21	64	51	(32)	34	1	3
4	8	21	32	51	64	(34)	1	2
5	8	21	32	34	64	(51)	1	1
6	8	21	32	34	51	64		

circled: selected
shaded: sorted subarray

#ex: no. of exchanges
#cm: no. of comparisons

Selection Sort: Code

```
void selectionsort(int n, int a[]){
    int i, j, min, tmp;
    for (i = 0; i < n - 1; i++){
        min = i;
        for (j = i + 1; j < n; j++){
            if (a[j] < a[min])
                min = j;
        }
        tmp = a[i];
        a[i] = a[min];
        a[min] = tmp;
    }
}
```

$N - 1$ passes

select the min.

exch.

Selection Sort: Analysis

- The algorithm consists of a selection phase in which the **smallest** of the remaining elements, ***min***, is repeatedly placed in its proper position, *i*.
- ***min*** is **swapped** with the element *a*[*i*].
- The initial *n*-element **priority queue** is reduced by one element after each selection.
- The first pass makes $N - 1$ comparisons, the second pass makes $N - 2$, and so on.
- Therefore there is a total of $(N - 1) + (N - 2) + \ldots + 1 = N(N - 1)/2 = O(N^2)$ comparisons.

Selection Sort: Analysis (2)

- The number of swaps is **always** (N - 1).
  - unless a test is added to prevent the swap of an element with itself.
- ☑ little additional storage required
  - only a few temporary variables
- The sort may be categorized as $O(n^2)$, although it is **faster** than the bubble sort.
- ☒ There is **NO** improvement if the input array is **completely sorted** or unsorted
  - The testing ignores the makeup of the array.

Selection Sort: Example 2

	H	E	L	L	O	W	O	R	L	D	#e	#c
1	H	E	L	L	O	W	O	R	L	D	1	9
2	D	E	L	L	O	W	O	R	L	H	1	8
3	D	E	L	L	O	W	O	R	L	H	1	7
4	D	E	H	L	O	W	O	R	L	L	1	6
5	D	E	H	L	O	W	O	R	L	L	1	5
6	D	E	H	L	L	W	O	R	O	L	1	4
7	D	E	H	L	L	L	O	R	O	W	1	3
8	D	E	H	L	L	L	O	R	O	W	1	2
9	D	E	H	L	L	L	O	O	R	W	1	1
10	D	E	H	L	L	L	O	O	R	W	0	0

Insertion Sort

- **Bridge hand** sorting
  - consider the items one at a time
  - insert each into its proper place among already considered(and sorted)
- In computer, we need to make **space** for moving larger items one position to the right.
- Clearly, insertion sort consists of *N* - 1 passes.
  - For pass *p* = 2 through *N*, insertion sort ensures that the elements in positions 1 through *p* are in sorted order.
  - Insertion sort makes use of the fact:
    Elements in positions 1 through *p* - 1 are already known to be in **sorted** order.

Insertion Sort: Example

	34	8	64	51	32	21	#ex	#cm
1	(8)	34	64	51	32	21	1	1
2	8	34	(64)	51	32	21	0	1
3	8	34	(51)	64	32	21	1	2
4	8	(32)	34	51	64	21	3	4
5	8	(21)	32	34	51	64	4	5

circled: selected
shaded: moved

#ex: no. of exchanges
#cm: no. of comparisons

Insertion Sort: Code

- Line 5-6: Shift down the elements until we find an element greater than $a[i]$.
- Line 7: insert the saved element at this point.

```
void insertionsort(int n, int a[]){
    int i, j, tmp;
    for (i = 1; i < n; i++){        N – 1 passes
        tmp = a[i];
        for (j = i; j > 0 && a[j - 1] > tmp; j--)      make
            a[j] = a[j - 1]; // shift down              space
        a[j] = tmp; // the insert
    }
}
```

Insertion Sort: Analysis (1)

- Initially $a[0]$ may be thought of as a sorted array of one element.
- After k iterations, the elements $a[0]$ through $a[k]$ are in **order**.
- **Time complexity** in various cases
  - If the initial array is sorted, only one comparison is made on each pass (line 5), so that the sort is $O(N)$.
  - If the array is initially sorted in the **reversed** order, the sort is $O(N^2)$, since the total number of comparisons is

$$(N - 1) + (N - 2) + \ldots + 2 + 1 = (N - 1)N/2 = O(N^2)$$

Insertion Sort: Analysis (2)

- ☑ The closer the array is to the sorted order, the more **efficient** the insertion sort becomes. (i.e. less **unnecessary** comparisons)
- The average number of comparisons in the simple insertion sort (by considering all possible permutations of the input array) is also $O(N^2)$.
- The space requirements consists of only one temporary variable ***tmp***.
- In worse case, insertion sort makes $O(N^2)$ comparisons of keys and $O(N^2)$ movements of entries.

Insertion Sort: Analysis (3)

- **Improve searching**: use a binary search.
  - This reduces the total number of comparisons from $O(N^2)$ to $O(N \log N)$.
  - The moving operation still requires $O(N^2)$ time. So the binary search **does not** significantly improves the overall time requirement.
- **Improve insertion**: use the **list insertion**.
  - This reduces the time required for insertion but not the time required for searching for the proper position.
  - Also this requires time and memory to build the linked list copy of the original array.

A Lower Bound for Simple Sorting

The average number of inversions in an array of N distinct numbers is $N(N - 1)/4$.

Proof

- For any list, L, of numbers, consider L_r , the list in **reversed** order.
- Consider any pair of two numbers in the set of keys (x, y), with $y > x$.
- In exactly one of L and L_r this ordered pair represents an **inversion**.
- The total number of these pairs is

$$(N - 1) + (N - 2) + \ldots + 1 = N(N - 1)/2$$

- On average, it is half of the above.

A Lower Bound for Simple Sorting (2)

Any algorithm that sorts by swapping adjacent elements requires $\Omega(N^2)$ time on average.

Proof

- The average number of inversions is initially $N(N - 1)/4 = \Theta(N^2)$
- Each swap removes **only** one inversion, so $\Theta(N^2)$ swaps are required.

Average case complexity of insertion sort

j loop executes $N(N - 1)/4$ times.

i loop executes exactly *n* times.

Complexity = $O(N) + N(N - 1)/4 = N^2/4 + O(N)$

Insertion Sort: Example 2

	H	E	L	L	O	W	O	R	L	D	#e	#c
1	E	H	L	L	O	W	O	R	L	D	1	1
2	E	H	L	L	O	W	O	R	L	D	0	1
3	E	H	L	L	O	W	O	R	L	D	0	1
4	E	H	L	L	O	W	O	R	L	D	0	1
5	E	H	L	L	O	W	O	R	L	D	0	1
6	E	H	L	L	O	O	W	R	L	D	1	2
7	E	H	L	L	O	O	R	W	L	D	1	2
8	E	H	L	L	L	O	O	R	W	D	4	5
9	D	E	H	L	L	L	O	O	R	W	9	10

Bubble Sort

- Scan the list from one end to the other, and whenever a pair of **adjacent** keys is found to be out of order, then those entries are **swapped**.
- In this pass, the largest key in the list will have **bubbled** to the end, but the earlier keys may still be out of order. (assume that you begin from the left hand side)
- The bubble sort is probably the easiest algorithm to implement but the **most time consuming** of all the algorithm.
- The basic idea underlying the bubble sort is to pass through the array sequentially several times.

Bubble Sort: Example

	34	8	64	51	32	21	#ex	#cm
1	8	34	51	32	21	64	4	5
2	8	34	32	21	51	64	2	4
3	8	32	21	34	51	64	2	3
4	8	21	32	34	51	64	1	2
5	8	21	32	34	51	64	0	1

(the bubbles)
circled: selected
shaded: sorted subarray

#ex: no. of exchanges
#cm: no. of comparisons

Bubble Sort: Code

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

```
void bubblesort(int n, int a[]) {
    int i, j, tmp;
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - 1 - i; j++)
            // compare the two neighbors
            if (a[j] > a[j + 1]) {
                // swap a[j] and a[j+1]
                tmp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = tmp;
            }
}
```

Bubble Sort: Improvement

- Stops when we spot that there is no more exchanges in the last round.

```
void bubblesort_imprv(int n, int a[]) {
    int i, j, tmp, swapped = 1;
    for (i = 0; i < n - 1 && swapped; i++){
        swapped = 0;
        for (j = 0; j < n - i - 1; j++){
            if (a[j] > a[j + 1]) {
                swapped = 1;
                tmp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = tmp;
            }
        }
    }
}
```

Bubble Sort: Analysis

- **Original** version:
  - The number of comparisons made in:
    1^{st} pass: $N - 1$; 2^{nd} pass: $N - 2$, and so on.
  - The total number of comparisons is
    $(N - 1) + (N - 2) + \ldots + 1 = N(N - 1)/2 = O(N^2)$$
- **With improvement**:
  - When it is close to sorted, $O(N)$
  - When it is far from sorted, $O(N^2)$
- ☑ Bubble sort requires little additional space.

Bubble Sort: Example 2

	H	E	L	L	O	W	O	R	L	D	#e	#c
1	E	H	L	L	O	O	R	L	D	W	5	9
2	E	H	L	L	O	O	L	D	R	W	2	8
3	E	H	L	L	O	L	D	O	R	W	2	7
4	E	H	L	L	L	D	O	O	R	W	2	6
5	E	H	L	L	D	L	O	O	R	W	1	5
6	E	H	L	D	L	L	O	O	R	W	1	4
7	E	H	D	L	L	L	O	O	R	W	1	3
8	E	D	H	L	L	L	O	O	R	W	1	2
9	D	E	H	L	L	L	O	O	R	W	1	1

Comparing Elementary Sorting Algo.

Algorithm	# cm	# ex
Bubble Sort	$N^2/2$	$N^2/2$
Insertion Sort	$N^2/4$ (avg.) $N^2/2$ (**worse**)	$N^2/4$ (avg.) $N^2/2$ (**worse**)
Selection Sort	$N^2/2$	N

Shell Sort

- Selection sort moves the entries very efficiently but does many redundant comparisons.
- Insertion sort uses the minimum number of comparisons, but is inefficient in moving element only one slot at a time.

Idea of Shell Sort

Modify the comparison method so that it first compares **distant** keys.

The distance between comparisons **decreases** as the algorithm runs through phases, with adjacent elements compared in the last phase.

Shell Sort

- **Increment Sequence**: $h_t, h_{t-1}, \ldots, h_2, h_1 = 1$,
- After a **phase** using some increment h_k, we have $a[i] \leq a[i + h_k]$ for all i (i.e. all elements spaced h_k apart is sorted).
- The array is said to be ***h_k*-sorted**.
- Shell sort is also called **diminishing increment sort**
  - since the increment sequence goes smaller and smaller (i.e. $h_{j+1} > h_j$)

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

Shell Sort: Strategy

- For a h_k-sort, for each position i in h_k, h_{k+1}, …, n - 1, place the element in the correct spot among i, i - h_k, i - $2h_k$, etc.
- You can imagine this is like **jumping** over h_k elements on the array.
- It is interesting that this is **equivalent** to perform an insertion sort on h_k **independent** subarrays.
- Example: $h_k = 3$

	6	8	13	2	5	11	20	9
1	6			2			20	
2		8			5			9
3			13			11		

Shell Sort: Code

- A **popular choice** for the increment sequence (as suggested by Shell): $h_t = \lfloor N/2 \rfloor \;\; h_k = \lfloor h_{k+1}/2 \rfloor$

```
void shellsort(int n, int a[]){
   int i, j, inc, tmp;
   for (inc = n / 2; inc > 0; inc /= 2)
       for (i = inc; i < n; i++){
           tmp = a[i];
           for (j = i; j >= inc; j -= inc)
              if (tmp < a[j - inc])
                  a[j] = a[j - inc];
              else
                 break;
           a[j] = tmp;
       }
}
```

we may use better sequence

Shell Sort: Upper Bound Analysis

The worse-case running time of Shell sort, using Shell's increments, is $\Theta(N^2)$.

Upper bound: a h_k-sort consists of h_k insertion sorts $O(N^2)$ of around N/h_k elements.

Total cost of a pass = $O(h_k(N/h_k)^2) = O(N^2/h_k)$

Summing over all passes,

$$O\left(\sum_{i=1}^{t} N^2/h_i\right) = O\left(N^2 \sum_{i=1}^{t} 1/h_i\right)$$

Shell's sequence is a geometric sequence with ratio = 2, $\sum_{i=1}^{t} 1/h_i < 2$

Thus the **worse** case complexity of shell sort = $O(N^2)$

Shell Sort: Lower Bound Analysis

- We pick $N = 2^x$ for some x, so that all increments are **even** except the last one $h_1 = 1$.
- We put $N/2$ largest numbers in the **even** positions and $N/2$ smallest numbers in the **odd** positions.
- Therefore in before the last phase, the $N/2$ largest numbers remain in even positions, similar for smallest numbers.
- The i-th smallest number is in $2i - 1$, to restore it requires $i - 1$ moves in the array.

$$\text{Total \# moves} = \sum_{i=1}^{N/2}(i-1) = \Omega(N^2)$$

After 4-sort	1	5	2	6	3	7	4	8
After 2-sort	1	5	2	6	3	7	4	8
After 1-sort	1	2	3	4	5	6	7	8

Shell Sort: Analysis (3)

- Although shell sort is **easy** to code, the analysis of its running time is **difficult** and the runtime **depends on the choice** of the increment sequence.
- An important theorem about Shell sort:
  - **An h_k-sorted array that is then h_{k-1}-sorted remains h_k-sorted.**
- Therefore it is clever to choose increments that are relatively prime, i.e. $\gcd(h_{i+1},h_i) = 1$ for $t > i \geq 1$
- **Hibbard's** increments: 1, 3, 7, …, $2^k - 1$ will give a worse-case running time $\Theta(n^{3/2})$.
- **Sedgewick's** increments: $9x4^i - 9x2^i + 1$ or $4^i - 3x2^i + 1$. Worst-case: $O(n^{4/3})$.

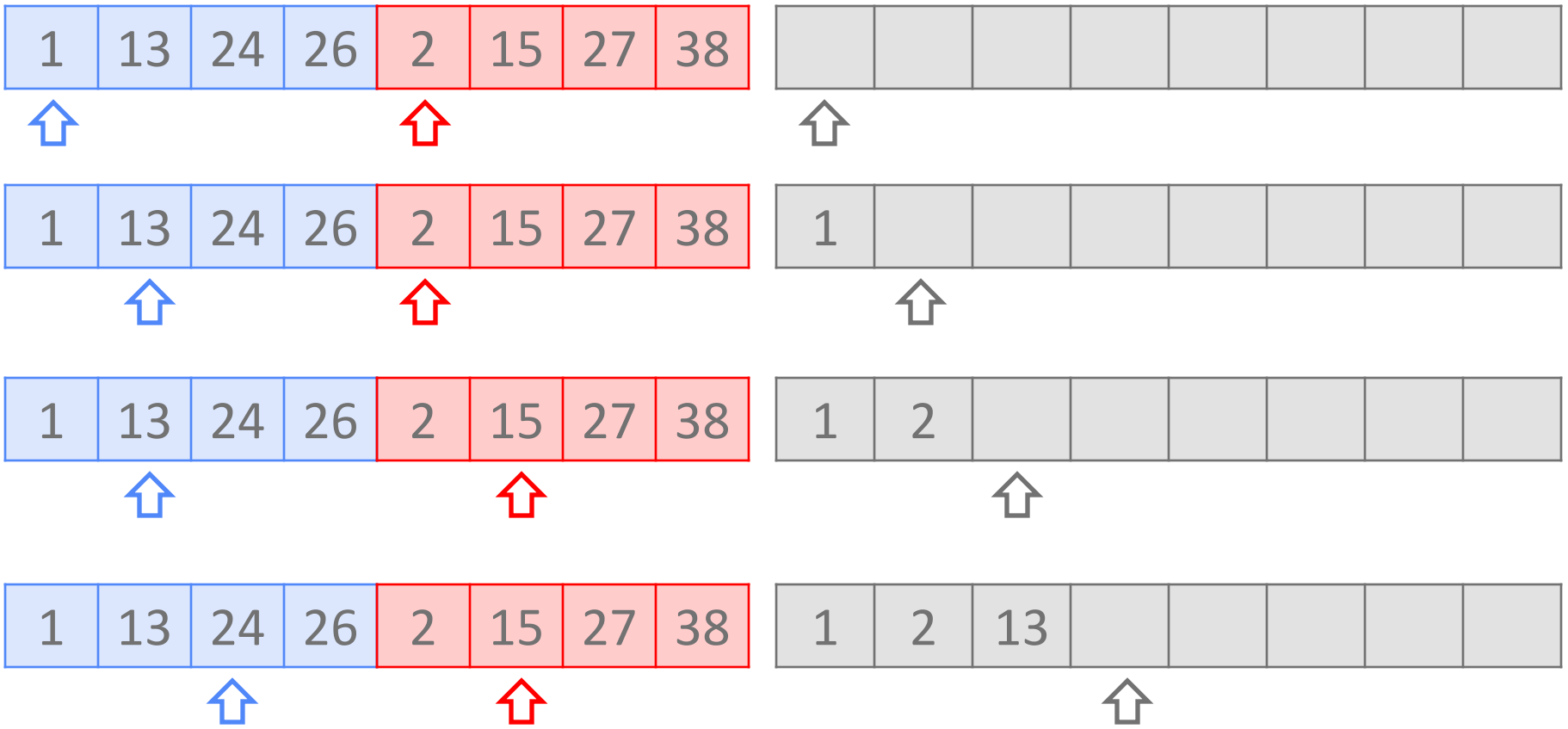
Part I: Summary

- Study several fundamental sorting algorithms based on swapping elements in list.
  - Insertion sort: minimum no. of **comparisons**
  - Selection sort: minimum no. of element **exchanges**
  - Bubble sort: the **slowest** algorithm
  - Shell sort: performance depends on choice of **increment sequence**.
- Understand the **lower** bound of $O(N^2)$ in sorting using swapping adjacent elements.

Merge sort

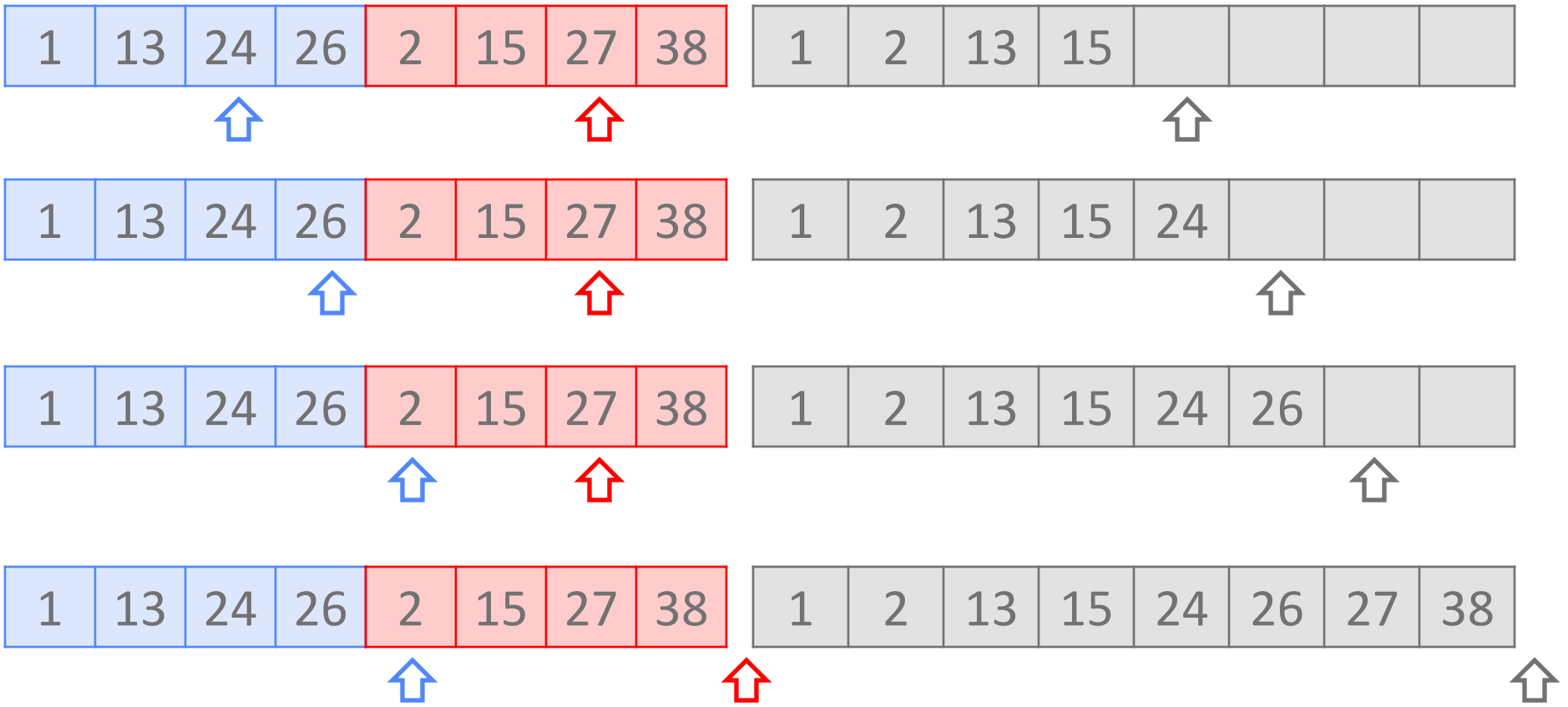
- The **classic** example of divide and conquer strategy:
  - **Divide**: Split the array into two halves and solve the sorting problem independently
  - **Conquer**: Merge the sorted sub-array to get back a whole sorted array.
- The fundamental operation is **merging** two **sorted** lists.
  - Since the lists (*A* and *B*) are sorted, the merging can be done by simply a linear scan through the input and put the output to an **auxiliary list** (*C*).
- Thus we use three counters (*a*, *b* and *c*) to record the current position on the three lists respectively.

Merging: Example



Is this merging process stable?

Merging: Example (2)



Merge Sort: Algorithm

- If $N = 1$, there is only one element to sort and we have the answer.
- Otherwise, recursively merge sort the first half (A) and the second half (B).
- The sorted two halves will be merged together using the procedure shown in the previous example.

Since the algorithm is recursive, we need a driver function to start the recursion. The function also allocates a temporary array.

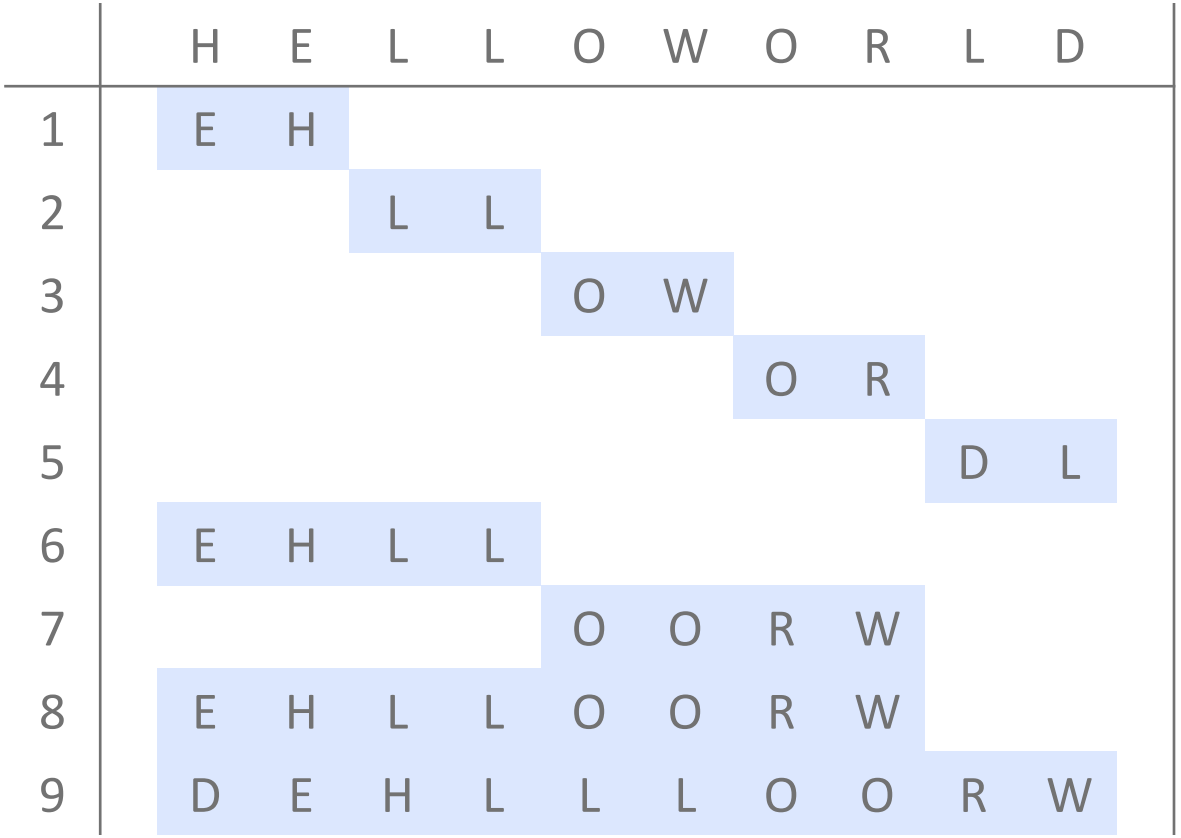
```
30 void mergesort(int n, int a[]){ /* driver */
31     int *tmpa = (int *)malloc(n * sizeof(int));
32
33     msort(a, tmpa, 0, n - 1);
34     free(tmpa);
35 }
```

Merge Sort: Example (1)

Top-down merge sort: Divide-and-conquer approach
Only merging steps in the recursion are shown.

	H	E	L	L	O	W	O	R	L	D
1	E	H								
2			L	L						
3			L	L	O					
4	E	H	L	L	O					
5						O	W			
6								L	R	
7								D	R	W
8						D	L	O	R	W
9	D	E	H	L	L	L	O	O	R	W

Merge Sort: Example (2)



	H	E	L	L	O	W	O	R	L	D
1	E	H								
2			L	L						
3					O	W				
4							O	R		
5									D	L
6	E	H	L	L						
7					O	O	R	W		
8	E	H	L	L	O	O	R	W		
9	D	E	H	L	L	L	O	O	R	W

Bottom-up merge sort: iterative, merge 1-1, then 2-2, 4-4, 8-8, etc.

Merge Sort: Code

- Case (i): right = left → 1 element; do nothing and return
- Case (ii): right = left + 1 → 2 elements;
  - center = left line 24 returns at once;
  - center + 1 = right line 25 returns at once.
- Then merge two single-entry sub-arrays.

```
19 void msort(int a[], int tmpa[], int left, int right){
20     int center;
21
22     if (left < right){
23         center = (left + right) / 2;
24         msort(a, tmpa, left, center);
25         msort(a, tmpa, center + 1, right);
26         merge(a, tmpa, left, center + 1, right);
27     }
28 }
```

Merge Sort: Code (2)

```
 1 void merge(int a[], int tmpa[], int l, int r, int rend){
 2     int i;
 3     int lend = r - 1;
 4     int t = l;
 5     int n = rend - l + 1;
 6
 7     while (l <= lend && r <= rend) // main loop
 8         tmpa[t++] = a[l] <= a[r] ? a[l++] : a[r++];
 9     while (l <= lend) // copy rest of first half
10         tmpa[t++] = a[l++];
11     while (r <= rend) // copy rest of second half
12         tmpa[t++] = a[r++];
13
14     // copy tmpa back
15     for (i = 0; i < n; i++, rend--)
16         a[rend] = tmpa[rend];
17 }
```

- Determine when the scan on either half is done: calculate *lend* and *rend*.

Merge Sort: Analysis

- For simplicity, we assume N is a power of 2.
  (The result remains **valid** when this assumption is violated)
- For $N = 1$, $T(N)$ is **constant** and we let $T(1) = 1$,
- otherwise, $T(N) = 2T(N/2) + N$,
  - we sort two halves and merge in linear time.
- To prepare for ***telescoping***, we divide the equation by N:

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

Merge Sort: Analysis (2)

- This is valid for any *N* that is a power of 2, so:

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

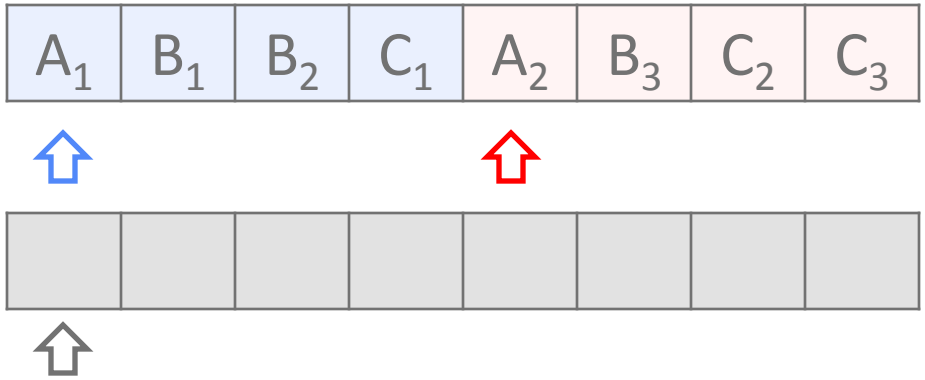
- Adding up all equations, we have

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \lg N$$
$$T(N) = N \lg N + N = O(N \lg N)$$

Properties of Merge Sort

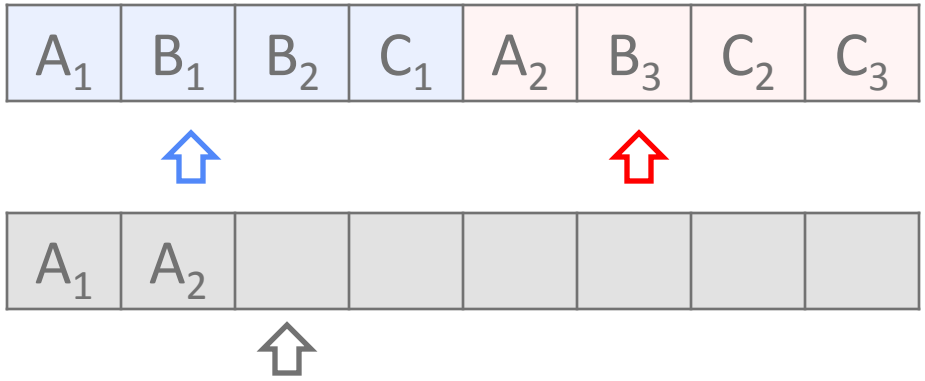
- ☑ Guaranteed to sort a file of N items in $O(N \lg N)$ time
- ☒ use up $O(N)$ extra space
  - need **careful** implementations for efficient sorting
- Running time depends **primarily** on only the number of input keys
  - insensitive of their relative orders
- **stable**

Stable Merging



Merge the subarrays into the aux array

Pay attention to the relative orders of the alphabets

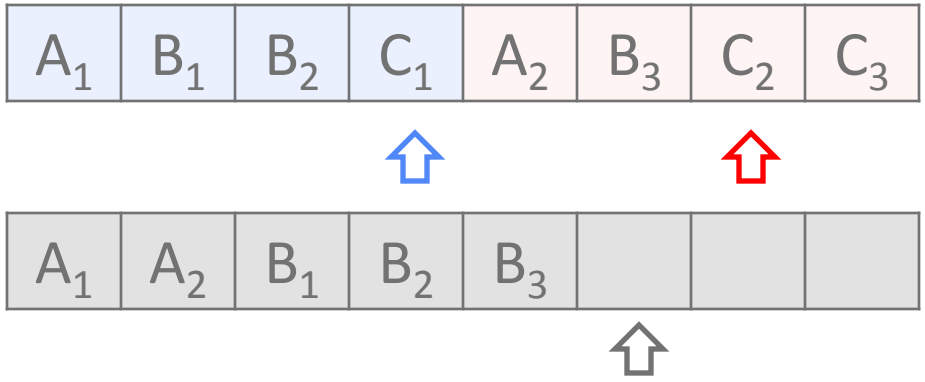


We always merge from the left array first.

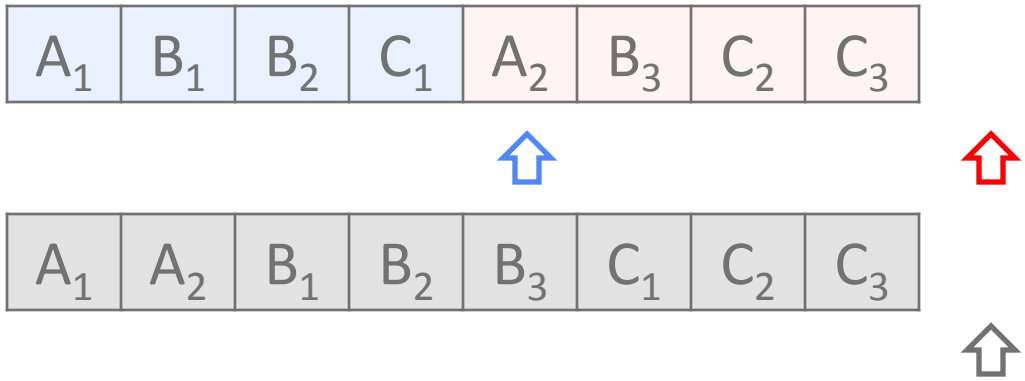
$key_1(A_1) = key_1(A_2)$

$key_2(A_1) < key_2(A_2)$

Stable Merging (2)



Continue with B_1, B_2, B_3



Merge sort is stable only when the merging step is stable and preserve relative orders of the items.

Quicksort

- ***quicksort*** is one of the **fastest** known sorting algorithms in **practice**.
- Its **worse-case** performance is $O(N^2)$ and its average runtime is $O(N \log N)$.
- A divide-and-conquer recursive algorithm.

Quicksort Algorithm

If the number of elements in S is 0 or 1, then return
Pick any element v in S, named the **pivot**.
Partition $S' = S - \{v\}$ into two disjoint groups:

$$S_1 = \{x \in S' | x \leq v\} \quad S_2 = \{x \in S' | x \geq v\}$$

Return $\{quicksort(S_1), v, quicksort(S_2)\}$

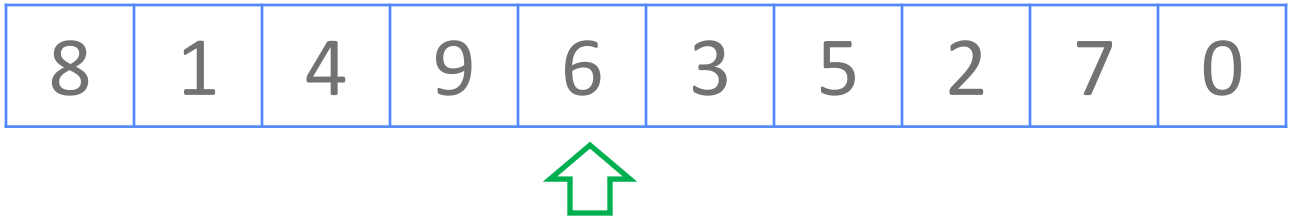
Quicksort: Pivot Selection

- Ideally we want half the keys to go into S_1 and another half into S_2.
  - The choice of pivot can **strongly** affect the relative sizes of the partitions.
- Why quicksort is faster than merge sort?
  - We can implement the partitioning to be performed in place and very efficiently. There is no need of an **extra** array in merge sort.

1. ☹ The first element: **bad** choice. Keep making bad partitioning when the array is sorted or reversed.
2. 😐 A random pivot: **safe**, but you take extra time to generate the random number.
3. ☺ Median-of-Three Partitioning: pick the median of $a[0]$, $a[n - 1]$ and $a[n / 2]$ to be the pivot.

Partitioning Strategy

Pick the pivot using median-of-three method



Swap the pivot with the last element (hiding)

Take two counters *i* and *j* at the first element and next-to-last element respectively.

Partitioning Strategy (2)

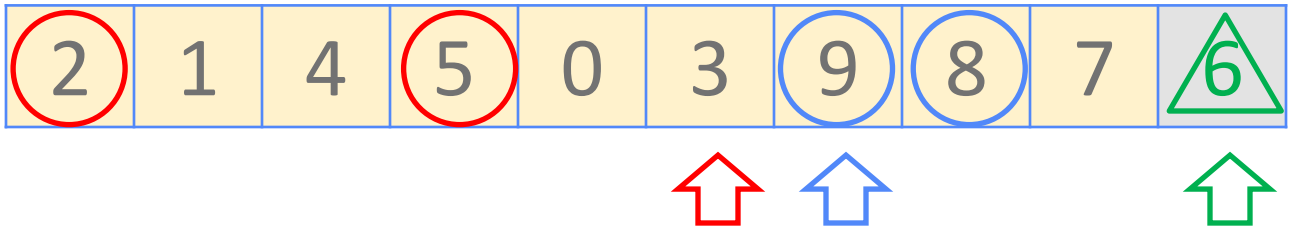
Goal: move all the small elements to the left part of the array while the large elements to the right part.

- move *i* until we find an element larger than the pivot
- move *j* until we find a smaller element
- exchange the two

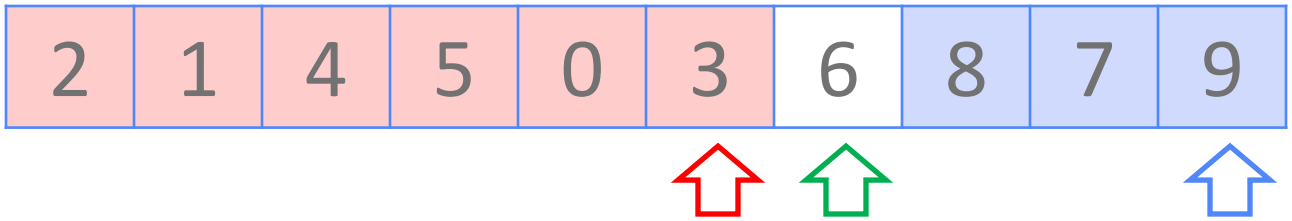
And repeat …

Partitioning Strategy (3)

Until *i* and *j* **cross**.



Swap the pivot with the element pointed by *i*. Done!



Quicksort: Code

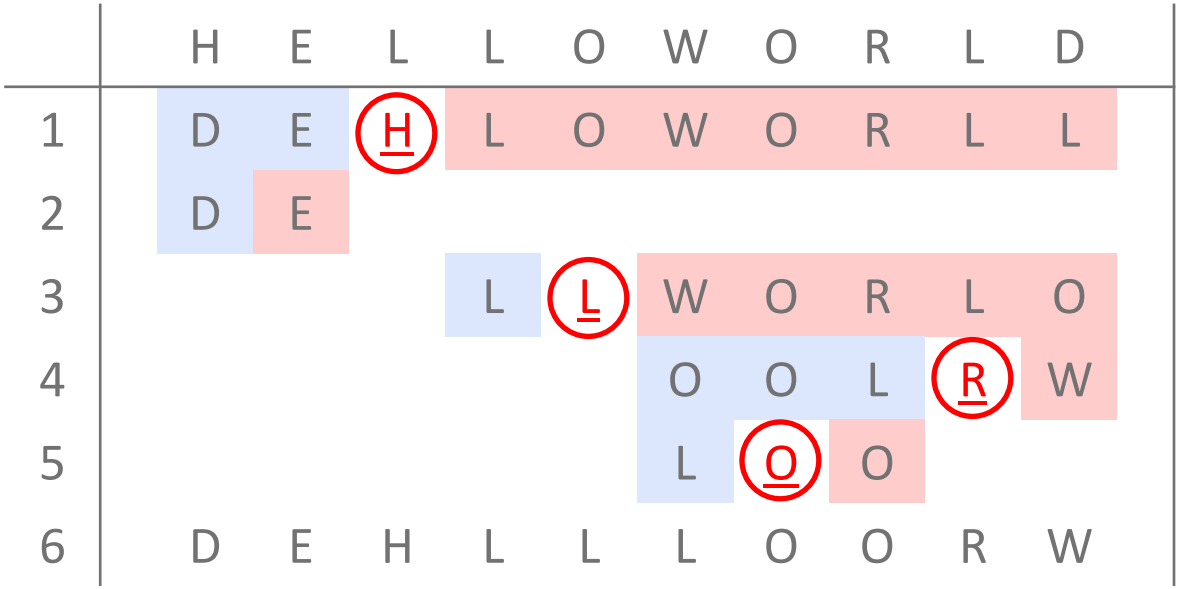
- The median-of-three pivot selection:

```
 9 int median3(int a[], int l, int r){
10     int c = (l + r) / 2;
11
12     if (a[l] > a[c]){
13         swap(&a[l], &a[c]);
14     if (a[l] > a[r])
15         swap(&a[l], &a[r]);
16     if (a[c] > a[r])
17         swap(&a[c], &a[r]);
18     // we are sure that:
19     // a[l] <= a[c] <= a[r]
20     swap(&a[c], &a[r]); // hide pivot
21     return a[r]; // return pivot
22 }
```

Quicksort: Code (2)

```
24 static void qsort_r(int *a, int left, int right){
25/6    int i, j, p; // handle simple cases
27      if (left >= right) return;
28      if (left + 1 == right){
29          if (a[left] > a[right])
30              swap(&a[left], &a[right]);
31          return;
32      }
33/4    p = median3(a, left, right); i = left; j = right;
35      while (1){
36          while (a[++i] < p);
37          while (a[--j] > p);
38          if (i < j)
39              swap(&a[i], &a[j]);
40          else
41              break;
42      } // restore pivot
43      swap(&a[i], &a[right]);
44/5    qsort_r(a, left, i - 1); qsort_r(a, i + 1, right);
46 }
```

Quicksort: Example



◯ : median-of-three pivot

Quicksort: Analysis

- When the array is small, it sorts in **constant** time: $T(0) = T(1) = 1$
- If we choose the pivot in constant time, the runtime for N elements is just
  - the time spent on **sorting** the two partitions +
  - the linear time spent in the **partitioning**:

$$T(N) = T(i) + T(N - i - 1) + cN$$

for some constant c, $i = |S_1|$ is the number of elements in S_1.

Quicksort: Worse-case Analysis

Worse case

We have pivot being the smallest elements throughout. Then $i = 0$. For $N > 1$,

$$T(N) = T(N-1) + cN$$
$$T(N-1) = T(N-2) + c(N-1)$$
$$T(N-2) = T(N-3) + c(N-2)$$
$$\vdots$$
$$T(2) = T(1) + c(2)$$

Adding up, we have

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

Quicksort: Best-case Analysis

Best case

The pivot is always in the **middle**. So the two subarrays are each close to half the size of the original.

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

Repeat the same technique we used in analyzing merge sort, we conclude that

$$T(N) = cN \lg N + N = O(N \lg N)$$

Quicksort: Avg-case Analysis

Average case

The average value of $T(i)$, and $T(n - i - 1)$, is $\frac{1}{N}\sum_{j=0}^{N-1} T(j)$

The runtime expression becomes:

$$T(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} T(j)\right] + cN \qquad (1)$$

$$NT(N) = 2\left[\sum_{j=0}^{N-1} T(j)\right] + cN^2 \qquad (2)$$

$$(N-1)T(N-1) = 2\left[\sum_{j=0}^{N-2} T(j)\right] + c(N-1)^2 \qquad (3)$$

Quicksort: Avg-case Analysis (2)

Subtract (3) from (2)

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c \rightarrow$$

$$NT(N) \approx (N+1)T(N-1) + 2cN$$ (dropping constant c)

Telescoping:

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

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

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2c}{N-1}$$

$$\vdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

Quicksort: Avg-case Analysis (3)

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

The harmonic number H_N appears, take approximation:

$$\frac{T(N)}{N+1} \approx \frac{T(1)}{2} + 2c \left(\ln(N+1) + \gamma - (1 + \frac{1}{2}) \right)$$

$$\frac{T(N)}{N+1} = O(\ln N)$$

$$T(N) = O(N \ln N)$$

A General Lower Bound for Sorting

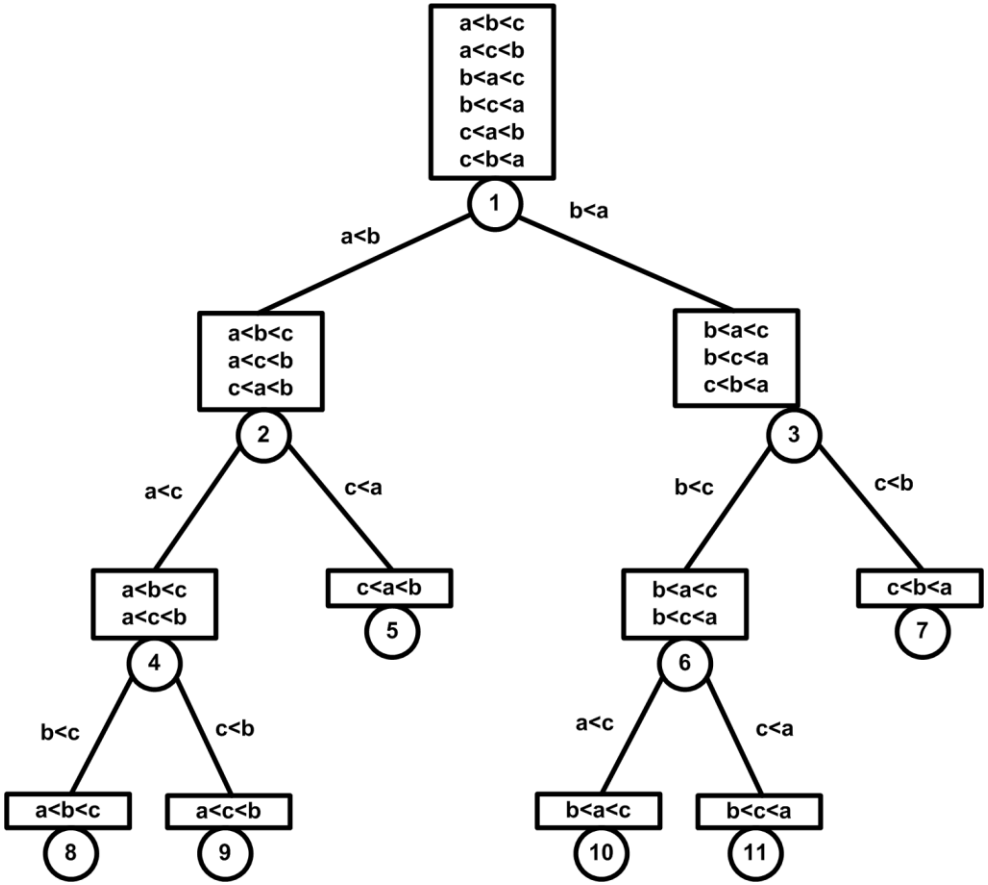
- How **fast** can we sort?
  - Bubble sort, insertion sort and selection sort: $O(N^2)$
  - Heapsort, merge sort and quicksort(avg.): $O(N \log N)$
- **Comparison** is the basic common technique that we have been using for sorting.
  - The number of comparisons made during the sort **dominates** the overall runtime of the algorithm.
- We are going to show that $\Omega(N \lg N)$ comparisons are always required ➔ a lower bound for general sorting algorithm using comparisons is $\Omega(N \lg N)$.

Decision Trees

- We answer **YES** or **NO** when we compare.
- Suppose we record the decisions of YES and NO in a binary tree: if you answer YES, add a **left** child; if you answer NO, add a **right** child.
- The resultant tree is called a **decision** tree.
- No matter in what order you compare, you need to **distinguish one permutation with another** so that you can confirm on the sorted and correct answer.
- Every algorithm that sorts by using only comparisons can be represented by a decision tree.

Decision Tree: Example

- Insertion sort on an array of 3 elements *a*, *b* and *c*.



Lower Bound for Sorting

- Recall that any binary tree T of height h has at most 2^h leaves.
- So a binary tree with L leaves must have **height** at least **ceil(lg L)**.
- A decision tree to sort N elements has **N!** leaves (to distinguish every permutation), so the height of the tree is **ceil(lg N!)**.

$$\lg (N!) > \lg (N/3)^N = N \lg (N/3) = \Omega(N \lg N)$$

- That means in the **worse** case we need at least (N lg N) comparisons to sort.

$(N/3)^n < N! < (N/2)^N$. For a tighter bound, use Stirling's approximation

Part II: Summary

- **Merge** sort: solve the sorting by divide-and-conquer strategy. Fast and stable but requires $O(N)$ memory.
- **Quicksort**: fast, efficient and sort in-place. Constrained by pivot selection and can be $O(N^2)$ in **worse** case.
- In general, we need $O(N \lg N)$ comparisons to distinguish every permutations of N elements.