Sorting Problem and Sorting Algorithms

1. Introduction

Sorting is a very classic problem of reordering items (that can be compared, e.g., integers, floating-point numbers, strings, etc) of an array (or a list) in a certain order (increasing, non-decreasing (increasing or flat), decreasing, non-increasing (decreasing or flat), lexicographical, etc).

There are many different sorting algorithms, each has its own advantages and limitations.

Sorting is commonly used as the introductory problem in various Computer Science classes to showcase a range of algorithmic ideas.

Without loss of generality, we assume that we will sort only **Integers**, not necessarily distinct, in **non-decreasing order** in this visualization. Try clicking Bubble Sort for a sample animation of sorting the list of 5 jumbled integers (with duplicate) above.

1-1. Motivation - Interesting CS Ideas

Sorting problem has a variety of interesting algorithmic solutions that embody many Computer Science ideas:

1. Comparison versus non-comparison based strategies,
2. Iterative versus Recursive implementation,
3. Divide-and-Conquer paradigm (e.g., Merge Sort or Quick Sort),
4. Best/Worst/Average-case Time Complexity analysis,
5. Randomized Algorithms, etc.

1-2. Motivation - Applications

When an (integer) array **A** is sorted, many problems involving **A** become easy (or easier):

1. Searching for a specific value **v** in array **A**,
2. Finding the min/max or the k-th smallest/largest value in (static) array **A**,
3. Testing for uniqueness and deleting duplicates in array **A**,
4. Counting how many time a specific value **v** appear in array **A**,
5. Set intersection/union between array **A** and another sorted array **B**,
6. Finding a target pair $\mathbf{x} \in \mathbf{A}$ and $\mathbf{y} \in \mathbf{A}$ such that $\mathbf{x}+\mathbf{y}$ equals to a target **z**,

7. Counting how many values in array **A** is inside range [**lo**..**hi**], etc.

Discussion: In real-life classes, the instructor may elaborate more on these applications.

1-3. Some Hints

[This is a hidden slide]

2. Actions

[This sentence is irrelevant for notes]

2-1. Define Your Own Input

The first action is about defining **your own** input, an array/a list **A** that is:

1. Totally random,
2. Random but sorted (in non-decreasing or non-increasing order),
3. Random but **nearly** sorted (in non-decreasing or non-increasing order),
4. Random and contain many duplicates (thus small range of integers), or
5. Defined solely by yourself.

[This sentence is irrelevant for notes]

2-2. Execute the Selected Sorting Algorithm

[This sentence is irrelevant for notes]

3. Visualisation

[This sentence is irrelevant for notes]

4. Common Sorting Algorithms

[This sentence is irrelevant for notes]

The first six algorithms in this module are **comparison-based** sorting algorithms while the last two are not. We will discuss this idea midway through this e-Lecture.

The middle three algorithms are **recursive** sorting algorithms while the rest are usually implemented iteratively.

4-1. Abbreviations

To save screen space, we abbreviate algorithm names into three characters each:

1. Comparison-based Sorting Algorithms:
    1. BUB - Bubble Sort,
    2. SEL - Selection Sort,
    3. INS - Insertion Sort,
    4. MER - Merge Sort (recursive implementation),
    5. QUI - Quick Sort (recursive implementation),
    6. R-Q - Random Quick Sort (recursive implementation).
2. Not Comparison-based Sorting Algorithms:
    1. COU - Counting Sort,
    2. RAD - Radix Sort.

5. 3 O(N^2) Comparison-based Sorting Algorithms

We will discuss three comparison-based sorting algorithms in the next few slides:

1. Bubble Sort,
2. Selection Sort,
3. Insertion Sort.

They are called **comparison-based** as they compare pairs of elements of the array and decide whether to swap them or not.

These three sorting algorithms are the easiest to implement but also not the most efficient, as they run in O($\mathbf{N}^2$).

6. Analysis of Algorithms (Basics)

Before we start with the discussion of various sorting algorithms, it may be a good idea to discuss the basics of asymptotic algorithm analysis, so that you can follow the discussions of the various O(**N**^2), O(**N** log **N**), and special O(**N**) sorting algorithms later.

This section can be skipped if you already know this topic.

6-1. Mathematical Pre-requisites

You need to already understand/remember all these:
-. Logarithm and Exponentiation, e.g., $\log_2(1024) = 10$, $2^{10} = 1024$
-. Arithmetic progression, e.g., 1+2+3+4+…+10 = 10*11/2 = 55
-. Geometric progression, e.g., $1+2+4+8+..+1024 = 1*(1-2^{11})/(1-2) = 2047$
-. Linear/Quadratic/Cubic function, e.g., $f1(x) = x+2$, $f2(x) = x^2+x-1$, $f3(x) = x^3+2x^2-x+7$
-. Ceiling, Floor, and Absolute function, e.g., ceil(3.1) = 4, floor(3.1) = 3, abs(-7) = 7

6-2. What Is It?

Analysis of Algorithm is a process to evaluate rigorously the resources (time and space) needed by an algorithm and represent the result of the evaluation with a (simple) formula.

The time/space requirement of an algorithm is also called the time/space complexity of the algorithm, respectively.

For this module, we focus more on time requirement of various sorting algorithms.

6-3. Measuring the Actual Running Time?

We can measure the actual running time of a program by using wall clock time or by inserting timing-measurement code into our program, e.g., see the code shown in SpeedTest.cpp | py | java.

However, actual running time is not meaningful when comparing two algorithms as they are possibly coded in different languages, using different data sets, or running on different computers.

6-4. Counting # of Operations (1)

Instead of measuring the actual timing, we count the # of operations (arithmetic, assignment, comparison, etc). This is a way to assess its efficiency as an algorithm's execution time is correlated to the # of operations that it requires.

See the code shown in SpeedTest.cpp | py | java and the comments (especially on how to get the final value of variable counter).

Knowing the (precise) number of operations required by the algorithm, we can state something like this: Algorithm **X** takes $\mathbf{2n^2 + 100n}$ operations to solve problem of size **n**.

6-5. Counting # of Operations (2)

If the time **t** needed for one operation is known, then we can state that algorithm **X** takes $\mathbf{(2n^2 + 100n)t}$ time units to solve problem of size **n**.

However, time **t** is dependent on the factors mentioned earlier, e.g., different languages, compilers and computers, etc.

Therefore, instead of tying the analysis to actual time **t**, we can state that algorithm **X** takes time that is **proportional to $\mathbf{2n^2 + 100n}$** to solving problem of size **n**.

6-6. Asymptotic Analysis

Asymptotic analysis is an analysis of algorithms that focuses on analyzing problems of **large input size n**, considers **only the leading term** of the formula, and **ignores the coefficient** of the leading term.

We choose the leading term because the lower order terms contribute lesser to the overall cost as the input grows larger, e.g., for $f(n) = 2n^2 + 100n$, we have:
$f(1000) = 2*1000^2 + 100*1000 = 2.1M$, vs
$f(100000) = 2*100000^2 + 100*100000 = 20010M$.
(notice that the lower order term 100n has lesser contribution).

6-7. Ignoring Coefficient of the Leading Term

Suppose two algorithms have $2n^2$ and $30n^2$ as the leading terms, respectively.

Although actual time will be different due to the different constants, the growth rates of the running time are the same.

Compared with another algorithm with leading term of n^3, the difference in growth rate is a much more dominating factor.

Hence, we can drop the coefficient of leading term when studying algorithm complexity.

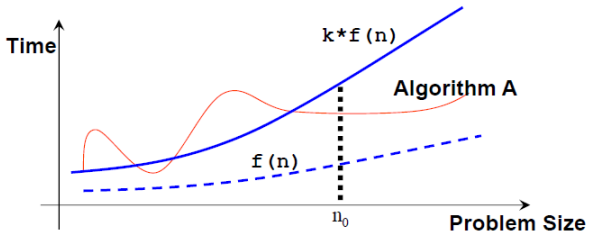
6-8. Upper Bound: The Big-O Notation

If algorithm A requires time proportional to **f(n)**, we say that algorithm A is of the order of f(n).

We write that algorithm A has time complexity of **O(f(n))**, where **f(n)** is the growth rate function for algorithm A.

6-9. Big-O Notation (Mathematics)

Mathematically, an algorithm A is of O(**f(n)**) if there exist a constant **k** and a positive integer **n0** such that algorithm A requires no more than **k*f(n)** time units to solve a problem of size **n ≥ n0**, i.e., when the problem size is larger than **n0**, then algorithm A is (always) bounded from above by this simple formula **k*f(n)**.



Note that: **n0** and **k** are not unique and there can be many possible valid **f(n)**.

6-10. Growth Terms

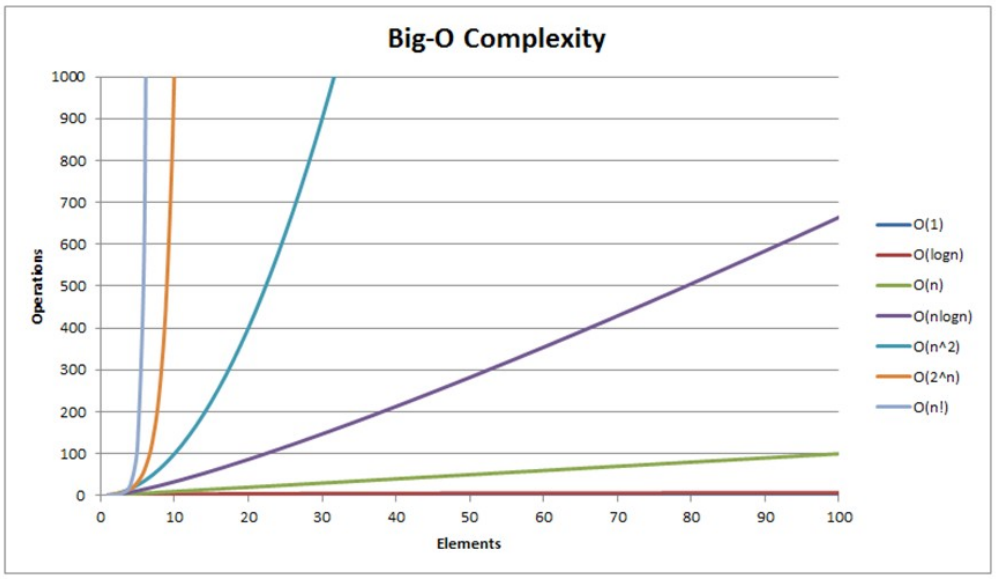
In asymptotic analysis, a formula can be simplified to a single term with coefficient 1.

Such a term is called a growth term (rate of growth, order of growth, order of magnitude).

The most common growth terms can be ordered from fastest to slowest as follows:
O(**1**)/constant time < O(log **n**)/logarithmic time < O(**n**)/linear time < O(**n** log **n**)/quasilinear time < O($\mathbf{n}^2$)/quadratic time < O($\mathbf{n}^3$)/cubic time < O($2^{\mathbf{n}}$)/exponential time < O(**n**!)/also-exponential time < ∞ (e.g., an infinite loop).

Note that a few other common time complexities are not shown (also see the visualization in the next slide).

6-11. Growth Terms (Visualized/Compared)



We will see three different growth rates O(**n**^{2}), O(**n log n**), and O(**n**) throughout the remainder of this sorting module.

7. Bubble Sort

Given an array of **N** elements, Bubble Sort will:

1. **Compare** a pair of adjacent items (a, b),
2. Swap that pair if the items are out of order (in this case, when a > b),
3. Repeat Step 1 and 2 until we reach the end of array
   (the last pair is the (**N**-2)-th and (**N**-1)-th items as we use 0-based indexing),
4. By now, the largest item will be at the last position.
   We then reduce **N** by 1 and repeat Step 1 until we have **N = 1**.

[This sentence is irrelevant for notes]

7-1. Bubble Sort, C++ Code & Analysis

```
void bubbleSort(int a[], int N) { // the standard version
  for (; N > 0; --N) // N iterations
    for (int i = 0; i < N-1; ++i) // except the last, O(N)
      if (a[i] > a[i+1]) // not in non-decreasing order
        swap(a[i], a[i+1]); // swap in O(1)
}
```

Comparison and swap require time that is bounded by a constant, let's call it **c**. Then, there are two nested loops in (the standard) Bubble Sort. The outer loop runs for exactly **N** iterations. But the inner loop runs get shorter and shorter:

1. When i=0, (**N**−1) iterations (of comparisons and possibly swaps),
2. When i=1, (**N**−2) iterations,
   ...,
3. When i=(**N**−2), 1 iteration,
4. When i=(**N**−1), 0 iteration.

Thus, the total number of iterations = (**N**−1)+(**N**−2)+...+1+0 = **N***(**N**−1)/2 (derivation).

Total time = c***N***(**N**−1)/2 = O(**N**^2).

7-2. Bubble Sort: Early Termination

Bubble Sort is actually inefficient with its **O(N^2)** time complexity. Imagine that we have **N** = 10^5 numbers. Even if our computer is super fast and can compute 10^8 operations in 1 second, Bubble Sort will need about 100 seconds to complete.

However, it can be terminated early, e.g. on the small sorted ascending example shown above [3, 6, 11, 25, 39], Bubble Sort can terminates in O(**N**) time.

The improvement idea is simple: If we go through the inner loop with **no swapping** at all, it means that the array is **already sorted** and we can stop Bubble Sort at that point.

Discussion: Although it makes Bubble Sort runs faster in general cases, this improvement idea does not change **O(N^2)** time complexity of Bubble Sort... Why?

7-3. The Answer

[This is a hidden slide]

8. Selection Sort

Given an array of **N** items and **L** = 0, Selection Sort will:

1. Find the position **X** of the smallest item in the range of [**L**...**N**−1],
2. Swap **X**-th item with the **L**-th item,
3. Increase the lower-bound **L** by 1 and repeat Step 1 until **L** = **N**-2.

[This sentence is irrelevant for notes]

Without loss of generality, we can also implement Selection Sort in reverse: Find the position of the largest item **Y** and swap it with the last item.

8-1. Selection Sort, C++ Code & Analysis

```
void selectionSort(int a[], int N) {
  for (int L = 0; L <= N-2; ++L) { // O(N)
    int X = min_element(a+L, a+N) - a; // O(N)
    swap(a[X], a[L]); // O(1), X may be equal to L (no actual swap)
  }
}
```

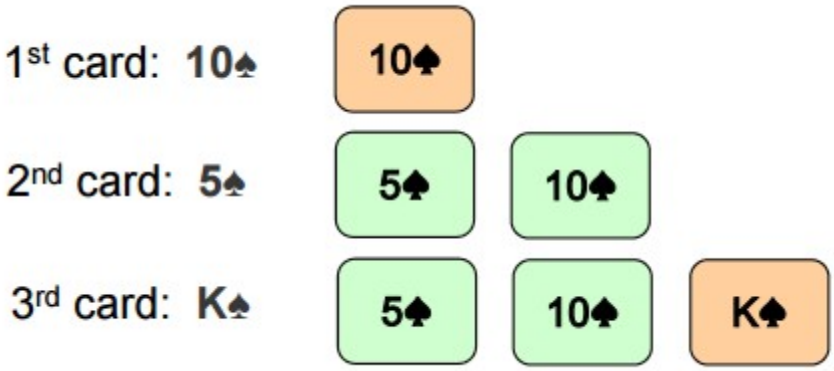
Total: $O(N^2)$ — To be precise, it is similar to Bubble Sort analysis.

8-2. Mini Quiz

[This sentence is irrelevant for notes]

9. Insertion Sort

Insertion sort is similar to how most people arrange a hand of poker



cards.

1. Start with one card in your hand,
2. Pick the next card and insert it into its proper sorted order,
3. Repeat previous step for all cards.

[This sentence is irrelevant for notes]

9-1. Insertion Sort, C++ Code and Analysis 1

```
void insertionSort(int a[], int N) {
  for (int i = 1; i < N; ++i) { // O(N)
    int X = a[i]; // X is the item to be inserted
    int j = i-1;
    for (; j >= 0 && a[j] > X; --j) // can be fast or slow
      a[j+1] = a[j]; // make a place for X
    a[j+1] = X; // index j+1 is the insertion point
  }
}
```

9-2. Insertion Sort: Analysis 2

The outer loop executes **N**−1 times, that's quite clear.

But the number of times the inner-loop is executed depends on the input:

1. In best-case scenario, the array is already sorted and (a[j] > X) is always false So no shifting of data is necessary and the inner loop runs in O(**1**),
2. In worst-case scenario, the array is reverse sorted and (a[j] > X) is always true Insertion always occur at the front of the array and the inner loop runs in O(**N**).

Thus, the best-case time is O(**N** × **1**) = O(**N**) and the worst-case time is O(**N** × **N**) = O($\mathbf{N}^2$).

9-3. Mini Quiz

[This sentence is irrelevant for notes]

10. 2.5 O(N log N) Comparison-based Sorting

We will discuss two (and a half) comparison-based sorting algorithms soon:

1. Merge Sort,
2. Quick Sort and its Randomized version (which only has one change).

These sorting algorithms are usually implemented recursively, use Divide and Conquer problem solving paradigm, and run in O(**N** log **N**) time for Merge Sort and O(**N** log **N**) time *in expectation* for Randomized Quick Sort.

PS: The non-randomized version of Quick Sort runs in O($\mathbf{N}^2$) though.

11. Merge Sort

Given an array of **N** items, Merge Sort will:

1. Merge each pair of individual element (which is by default, sorted) into sorted arrays of 2 elements,
2. Merge each pair of sorted arrays of 2 elements into sorted arrays of 4 elements, Repeat the process...,
3. Final step: Merge 2 sorted arrays of **N**/2 elements (for simplicity of this discussion, we assume that **N** is even) to obtain a fully sorted array of **N** elements.

This is just the general idea and we need a few more details before we can discuss the true form of Merge Sort.

11-1. Important Subroutine, O(N) Merge

We will dissect this Merge Sort algorithm by first discussing its most important sub-routine: The O(**N**) `merge`.

Given two sorted array, A and B, of size $\mathbf{N_1}$ and $\mathbf{N_2}$, we can efficiently merge them into one larger combined sorted array of size $\mathbf{N} = \mathbf{N_1}+\mathbf{N_2}$, in O(**N**) time.

This is achieved by simply comparing the front of the two arrays and take the smaller of the two at all times. However, this simple but fast O(**N**) `merge` sub-routine will need additional array to do this merging correctly.

11-2. Merge Subroutine C++ Implementation

```
void merge(int a[], int low, int mid, int high) {
  // subarray1 = a[low..mid], subarray2 = a[mid+1..high], both sorted
  int N = high-low+1;
  int b[N]; // discuss: why do we need a temporary array b?
  int left = low, right = mid+1, bIdx = 0;
  while (left <= mid && right <= high) // the merging
    b[bIdx++] = (a[left] <= a[right]) ? a[left++] : a[right++];
  while (left <= mid) b[bIdx++] = a[left++]; // leftover, if any
  while (right <= high) b[bIdx++] = a[right++]; // leftover, if any
  for (int k = 0; k < N; ++k) a[low+k] = b[k]; // copy back
}
```

[This sentence is irrelevant for notes]

11-3. Divide and Conquer Paradigm

Before we continue, let's talk about Divide and Conquer (abbreviated as D&C), a powerful problem solving paradigm.

Divide and Conquer algorithm solves (certain kind of) problem — like our sorting problem — in the following steps:

1. Divide step: Divide the large, original problem into smaller sub-problems and recursively solve the smaller sub-problems,
2. Conquer step: Combine the results of the smaller sub-problems to produce the result of the larger, original problem.

11-4. Merge Sort as a D&C Algorithm

Merge Sort is a Divide and Conquer sorting algorithm.

The divide step is simple: Divide the current array into two halves (perfectly equal if **N** is even or one side is slightly greater by one element if **N** is odd) and then recursively sort the two halves.

The conquer step is the one that does the most work: Merge the two (sorted) halves to form a sorted array, using the merge sub-routine discussed earlier.

11-5. Merge Sort C++ Implementation

```
void mergeSort(int a[], int low, int high) {
  // the array to be sorted is a[low..high]
  if (low < high) { // base case: low >= high (0 or 1 item)
    int mid = (low+high) / 2;
    mergeSort(a, low  , mid ); // divide into two halves
    mergeSort(a, mid+1, high); // then recursively sort them
    merge(a, low, mid, high); // conquer: the merge subroutine
  }
}
```

11-6. Demonstration

Contrary to what many other CS printed textbooks usually show (as textbooks are static), the actual execution of Merge Sort does **not** split to two subarrays **level by level**, but it will recursively sort the **left** subarray first before dealing with the **right** subarray.

That's it, running Merge Sort on the example array [7, 2, 6, 3, 8, 4, 5], it will recurse to [7, 2, 6, 3], then [7, 2], then [7] (a single element, sorted by default), backtrack, recurse to [2] (sorted), backtrack, then finally merge [7, 2] into [2, 7], before it continue processing [6, 3] and so on.

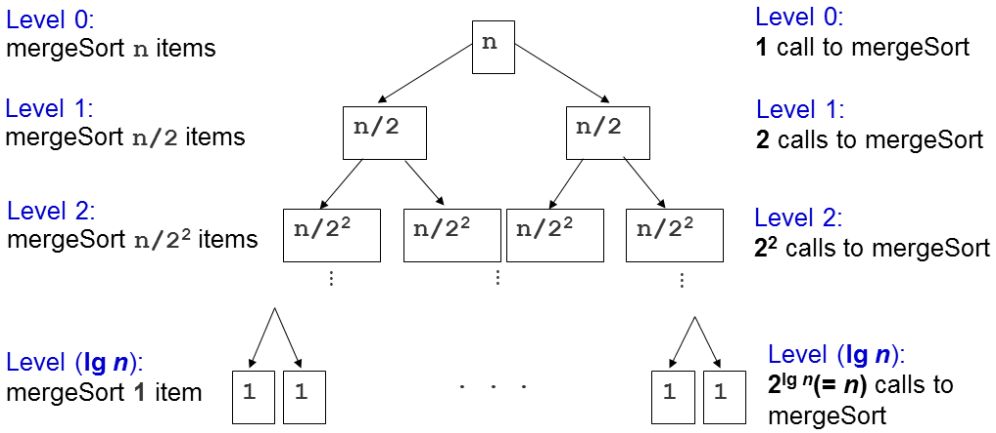
11-7. Merge Sort: Analysis Part 1

In Merge Sort, the bulk of work is done in the conquer/merge step as the divide step does not really do anything (treated as O(**1**)).

When we call `merge(a, low, mid, high)`, we process **k = (high-low+1)** items.
There will be at most **k-1** comparisons.
There are **k** moves from original array **a** to temporary array **b** and another **k** moves back.
In total, number of operations inside `merge` sub-routine is < 3**k**-1 = O(**k**).

The important question is how many times this `merge` sub-routine is called?

11-8. Merge Sort: Analysis Part 2



$n/(2^k) = 1 \rightarrow n = 2^k \rightarrow k = \lg n$

11-9. Merge Sort: Analysis Part 3

Level 1: 2^0=1 calls to merge() with **N**/2^1 items each, O(2^0 x 2 x **N**/2^1) = O(**N**)
Level 2: 2^1=2 calls to merge() with **N**/2^2 items each, O(2^1 x 2 x **N**/2^2) = O(**N**)
Level 3: 2^2=4 calls to merge() with **N**/2^3 items each, O(2^2 x 2 x **N**/2^3) = O(**N**)
...
Level (log **N**): 2^(log **N**-1) (or **N**/2) calls to merge() with **N**/2^log **N** (or 1) item each, O(**N**)

There are log **N** levels and in each level, we perform O(**N**) work, thus the overall time complexity is O(**N** log **N**). We will later see that this is an optimal (comparison-based) sorting algorithm, i.e., we cannot do better than this.

11-10. Pros and Cons

The most important good part of Merge Sort is its O(**N** log **N**) performance guarantee, regardless of the original ordering of the input. That's it, there is **no** adversary test case that can make Merge Sort runs longer than O(**N** log **N**) for **any** array of **N** elements.

Merge Sort is therefore very suitable to sort extremely large number of inputs as O(**N** log **N**) grows much slower than the O($\mathbf{N}^2$) sorting algorithms that we have discussed earlier.

There are however, several not-so-good parts of Merge Sort. First, it is actually not easy to implement from scratch (but we don't have to). Second, it requires additional O(**N**) storage during merging operation, thus not really memory efficient and not in-place. Btw, if you are interested to see what have been done to address these (classic) Merge Sort not-so-good parts, you can read this.

Merge Sort is also a stable sort algorithm. Discussion: Why?

11-11. The Answer

[This is a hidden slide]

12. Quick Sort

Quick Sort is another Divide and Conquer sorting algorithm (the other one discussed in this visualization page is Merge Sort).

We will see that this deterministic, non randomized version of Quick Sort can have bad time complexity of O($\mathbf{N}^2$) on adversary input before continuing with the randomized and usable version later.

12-1. Quick Sort as a D&C Algorithm

Divide step: Choose an item **p** (known as the pivot)
Then partition the items of **a[i..j]** into three parts: **a[i..m-1]**, **a[m]**, and **a[m+1..j]**.
a[i..m-1] (possibly empty) contains items that are smaller than (or equal to) **p**.
a[m] = p, i.e., index **m** is the correct position for **p** in the sorted order of array **a**.
a[m+1..j] (possibly empty) contains items that are greater than (or equal to) **p**.
Then, recursively sort the two parts.

Conquer step: Don't be surprised... We do nothing :O!

If you compare this with Merge Sort, you will see that Quick Sort D&C steps are totally opposite with Merge Sort.

12-2. Important Sub-routine, O(N) Partition

We will dissect this Quick Sort algorithm by first discussing its most important sub-routine: The O(**N**) `partition` (classic version).

To partition **a[i..j]**, we first choose **a[i]** as the pivot **p**.

The remaining items (i.e., **a[i+1..j]**) are divided into 3 regions:

1. **S1** = **a[i+1..m]** where items are ≤ **p**,
2. **S2** = **a[m+1..k-1]** where items are ≥ **p**, and
3. Unknown = **a[k..j]**, where items are yet to be assigned to either **S1** or **S2**.

Discussion: Why do we choose **p** = **a[i]**? Are there other choices?

Harder Discussion: If **a[k] == p**, should we put it in region S1 or S2?

12-3. The Answer

[This is a hidden slide]

12-4. Partition - Continued

Initially, both **S1** and **S2** regions are empty, i.e., all items excluding the designated pivot **p** are in the unknown region.

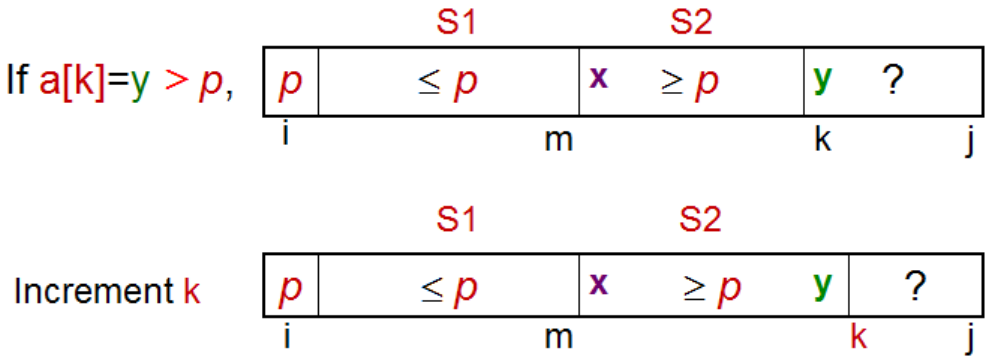
Then, for each item **a[k]** in the unknown region, we compare **a[k]** with **p** and decide one of the three cases:

1. If **a[k]** > **p**, put **a[k]** into **S2**,
2. If **a[k]** < **p**, put **a[k]** into **S1**,
3. If **a[k]** == **p**, throw a coin and put **a[k]** into **S1**/**S2** if it lands head/tail, respectively.

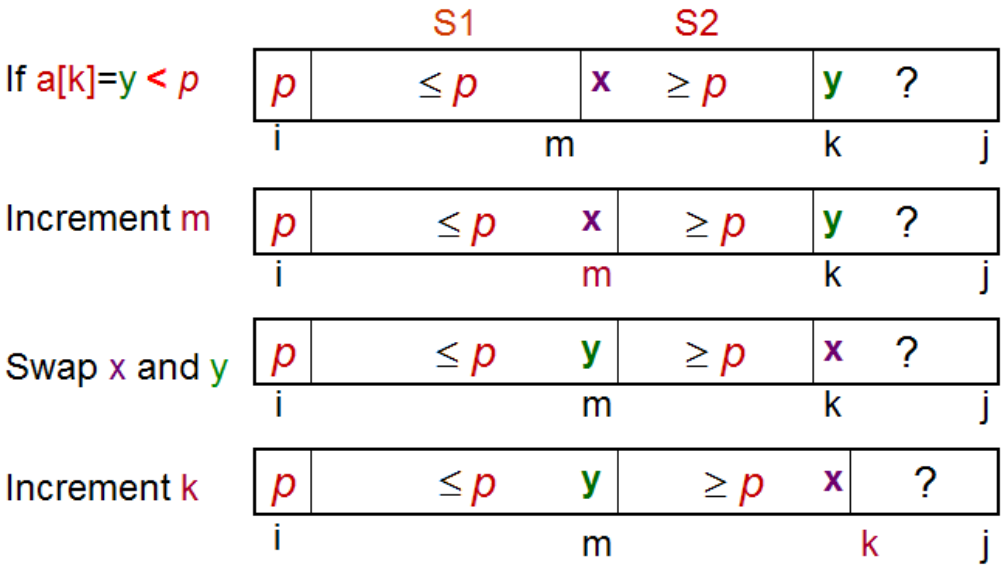
These three cases are elaborated in the next two slides.

Lastly, we swap **a[i]** and **a[m]** to put pivot **p** right in the middle of **S1** and **S2**.

12-5. Partition - Case when a[k] > p



12-6. Partition - Case when a[k] < p



12-7. Partition C++ Implementation

```
int partition(int a[], int i, int j) {
  int p = a[i]; // p is the pivot
  int m = i; // S1 and S2 are initially empty
  for (int k = i+1; k <= j; ++k) { // explore the unknown region
    if ((a[k] < p) || ((a[k] == p) && (rand()%2 == 0)))  { // case 2+3
      ++m;
      swap(a[k], a[m]); // C++ STL algorithm std::swap
    } // notice that we do nothing in case 1: a[k] > p
  }
  swap(a[i], a[m]); // final step, swap pivot with a[m]
  return m; // return the index of pivot
}
```

12-8. Quick Sort C++ Implementation

```
void quickSort(int a[], int low, int high) {
  if (low < high) {
```

```
    int m = partition(a, low, high); // O(N)
    // a[low..high] ~> a[low..m-1], pivot, a[m+1..high]
    quickSort(a, low, m-1); // recursively sort left subarray
    // a[m] = pivot is already sorted after partition
    quickSort(a, m+1, high); // then sort right subarray
  }
}
```

12-9. Demonstration

Try Quick Sort on example array [27, 38, 12, 39, 29, 16]. We shall elaborate the first partition step as follows:
We set **p = a[0] = 27**.
We set **a[1] = 38** as part of **S2** so **S1 = {}** and **S2 = {38}**.
We swap **a[1] = 38** with **a[2] = 12** so **S1 = {12}** and **S2 = {38}**.
We set **a[3] = 39** and later **a[4] = 29** as part of **S2** so **S1 = {12}** and **S2 = {38,39,29}**.
We swap **a[2] = 38** with **a[5] = 16** so **S1 = {12,16}** and **S2 = {39,29,38}**.
We swap **p = a[0] = 27** with **a[2] = 16** so **S1 = {16,12}**, **p = {27}**, and **S2 = {39,29,38}**.

After this, **a[2] = 27** is guaranteed to be sorted and now Quick Sort recursively sorts the left side **a[0..1]** first and later recursively sorts the right side **a[3..5]**.

12-10. Quick Sort: Analysis Part 1

First, we analyze the cost of one call of `partition`.

Inside `partition(a, i, j)`, there is only a single for-loop that iterates through (j-i) times. As j can be as big as **N**-1 and i can be as low as 0, then the time complexity of partition is O(**N**).

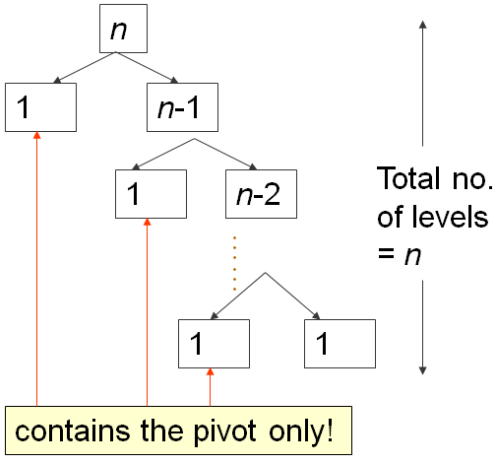
Similar to Merge Sort analysis, the time complexity of Quick Sort is then dependent on the number of times `partition(a, i, j)` is called.

12-11. Quick Sort: Analysis Part 2

When the array **a** is already in ascending order, e.g., **a** = [5, 18, 23, 39, 44, 50], Quick Sort will set **p = a[0] = 5**, and will return **m = 0**, thereby
making **S1** region **empty** and **S2** region: Everything else other than the pivot (**N**-1 items).

12-12. Quick Sort: Analysis Part 3

On such worst case input scenario, this is what happens:



The first partition takes O(**N**) time, splits **a** into 0, 1, **N**-1 items, then recurse right.
The second one takes O(**N**-1) time, splits **a** into 0, 1, **N**-2 items, then recurse right again.
...
Until the last, **N**-th partition splits **a** into 0, 1, 1 item, and Quick Sort recursion stops.

This is the classic **N+(N-1)+(N-2)+...+1** pattern, which is O($\mathbf{N}^2$), similar analysis as the one in this Bubble Sort analysis slide...

12-13. Quick Sort: Best Case (Rare)

The best case scenario of Quick Sort occurs when partition always splits the array into **two equal halves**, like Merge Sort.

When that happens, the depth of recursion is only O(log **N**).

As each level takes O(**N**) comparisons, the time complexity is O(**N** log **N**).

Try Quick Sort on this hand-crafted example input array [4, 1, 3, 2, 6, 5, 7].
In practice, this is rare, thus we need to devise a better way: Randomized Quick Sort.

13. Random Quick Sort

Same as **Quick Sort** except just before executing the partition algorithm,
it **randomly** select the pivot between **a[i..j]** instead of always choosing **a[i]** (or any other fixed index between **[i..j]**) deterministically.

Mini exercise: Implement the idea above to the implementation shown in this slide!

Running Random Quick Sort on this large and somewhat random example array **a** = [3,44,38,5,47,15,36,26,27,2,46,4,19,50,48] feels fast.

13-1. Magical Analysis

It will take about 1 hour lecture to properly explain why this randomized version of Quick Sort has expected time complexity of O(**N** log **N**) on **any** input array of **N** elements.

In this e-Lecture, we will assume that it is true.

If you need non formal explanation: Just imagine that on randomized version of Quick Sort that randomizes the pivot selection, we will **not** always get extremely bad split of 0 (empty), 1 (pivot), and **N**-1 other items. This combination of lucky (half-pivot-half), somewhat lucky, somewhat unlucky, and extremely unlucky (empty, pivot, the rest) yields an average time complexity of O(**N** log **N**).

Discussion: For the implementation of Partition, what happen if **a[k] == p**, we *always* put **a[k]** on either side (**S1** or **S2**) deterministically?

13-2. The Answer

[This is a hidden slide]

14. 2 O(N) Non Comparison-based Sorting Algorithms

We will discuss two **non comparison-based** sorting algorithms in the next few slides:

1. Counting Sort,
2. Radix Sort.

These sorting algorithms can be faster than the lower bound of comparison-based sorting algorithm of Ω(**N** log **N**) by **not** comparing the items of the array.

14-1. Lower Bound of Sorting Algorithm

It is known (also not proven in this visualization as it will take another 1 hour lecture to do so) that all **comparison-based** sorting algorithms have a lower bound time complexity of Ω(**N** log **N**).

Thus, any comparison-based sorting algorithm with worst-case complexity O(**N** log **N**), like Merge Sort is considered an optimal algorithm, i.e., we cannot do better than that.

However, we can achieve faster sorting algorithm — i.e., in O(**N**) — if certain assumptions of the input array exist and thus we can avoid comparing the items to determine the sorted order.

15. Counting Sort

Assumption: If the items to be sorted are **Integers with small range**, we can count the frequency of occurrence of each Integer (in that small range) and then loop through that small range to output the items in sorted order.

Try Counting Sort on the example array above where all Integers are within [1..9], thus we just need to count how many times Integer 1 appears, Integer 2 appears, ..., Integer 9 appears, and then loop through 1 to 9 to print out **x** copies of Integer **y** if frequency[**y**] = **x**.

The time complexity is O(**N**) to count the frequencies and O(**N+k**) to print out the output in sorted order where **k** is the range of the input Integers, which is 9-1+1 = 9 in this example. The time complexity of Counting Sort is thus O(**N+k**), which is O(**N**) if **k** is small.

We will not be able to do the counting part of Counting Sort when **k** is relatively big due to memory limitation, as we need to store frequencies of those **k** integers.

16. Radix Sort

Assumption: If the items to be sorted are **Integers with large range but of few digits**, we can combine Counting Sort idea with Radix Sort to achieve the linear time complexity.

In Radix Sort, we treat each item to be sorted as a string of **w** digits (we pad Integers that have less than **w** digits with leading zeroes if necessary).

For the least significant (rightmost) digit to the most significant digit (leftmost), we pass through the **N** items and put them according to the active digit into 10 Queues (one for each digit [0..9]), which is like a *modified* Counting Sort as this one preserves stability. Then we re-concatenate the groups again for subsequent iteration.

Try Radix Sort on the example array above for clearer explanation.

Notice that we only perform O(**w** × **(N+k)**) iterations. In this example, **w = 4** and **k = 10**.

16-1. The Best Sorting Algorithm for Integers?

Now, having discussed about Radix Sort, should we use it for **every** sorting situation?

For example, it should be theoretically faster to sort many (**N** is very large) 32-bit signed integers as **w ≤ 10** digits and **k = 10** if we interpret those 32-bit signed integers in Decimal. O(10 × (**N**+10)) = O(**N**).

16-2. The Answer

[This is a hidden slide]

17. Additional Properties of Sorting Algorithms

There are a few other properties that can be used to differentiate sorting algorithms on top of whether they are comparison or non-comparison, recursive or iterative.

In this section, we will talk about in-place versus not in-place, stable versus not stable, and caching performance of sorting algorithms.

17-1. In-Place Sorting

A sorting algorithm is said to be an **in-place sorting** algorithm if it requires only a constant amount (i.e. O(**1**)) of extra space during the sorting process. That's it, a few, constant number of extra variables is OK but we are not allowed to have variables that has variable length depending on the input size **N**.

Merge Sort (the classic version), due to its `merge` sub-routine that requires additional temporary array of size **N**, is not in-place.

Discussion: How about Bubble Sort, Selection Sort, Insertion Sort, Quick Sort (randomized or not), Counting Sort, and Radix Sort. Which ones are in-place?

17-2. Stable Sort

A sorting algorithm is called **stable** if the relative order of elements **with the same key value** is preserved by the algorithm after sorting is performed.

Example application of stable sort: Assume that we have student names that have been sorted in alphabetical order. Now, if this list is sorted again by tutorial group number (recall that one tutorial group usually has many students), a stable sort algorithm would ensure that all students in the same tutorial group still appear in alphabetical order of their names.

Discussion: Which of the sorting algorithms discussed in this e-Lecture are stable? Try sorting array A = {3, 4a, 2, 4b, 1}, i.e. there are two copies of 4 (4a first, then 4b).

17-3. Caching Performance

[This is a hidden slide]

18. Quizzes

We are nearing the end of this e-Lecture.

Time for a few simple questions.

18-1. Quiz #1

Quiz: **Which of these algorithms run in O(N log N) on any input array of size N?**

- ○ Insertion Sort
- ○ Quick Sort (Deterministic)
- ○ Merge Sort
- ○ Bubble Sort

18-2. Quiz #2

Quiz: **Which of these algorithms has worst case time complexity of Θ(N^2) for sorting N integers?**

- ☐ Bubble Sort
- ☐ Selection Sort
- ☐ Radix Sort
- ☐ Merge Sort
- ☐ Insertion Sort

Θ is a tight time complexity analysis where the best case Ω and the worst case big-O analysis match.

19. Extras

We have reached the end of sorting e-Lecture.

However, there are two other sorting algorithms in VisuAlgo that are embedded in other data structures: Heap Sort and Balanced BST Sort. We will discuss them when you go through the e-Lecture of those two data structures.

19-1. Challenge

[This is a hidden slide]

19-2. Inversion Index/Count

[This is a hidden slide]

19-3. Implementation

Actually, the C++ source code for many of these basic sorting algorithms are already scattered throughout these e-Lecture slides. For other programming languages, you can translate the given C++ source code to the other programming language.

Usually, sorting is just a small part in problem solving process and nowadays, most of programming languages have their own sorting functions so we don't really have to re-code them *unless absolutely necessary*.

In C++, you can use std::sort (most likely a hybrid sorting algorithm: Introsort), std::stable_sort (most likely Merge Sort), or std::partial_sort (most likely Binary Heap) in STL algorithm.
In Python, you can use sort (most likely a hybrid sorting algorithm: Timsort).
In Java, you can use Collections.sort.
In OCaml, you can use List.sort compare list_name.

If the comparison function is problem-specific, we may need to supply additional comparison function to those built-in sorting routines.

19-4. Online Quiz

Now it is time for you to see if you have understand the basics of various sorting algorithms discussed so far.

Test your understanding here!

19-5. Online Judge Exercises

Now that you have reached the end of this e-Lecture, do you think sorting problem is just as simple as calling built-in sort routine?

Try these online judge problems to find out more:
Kattis - mjehuric
Kattis - sortofsorting, or
Kattis - sidewayssorting

This is not the end of the topic of sorting. When you explore other topics in VisuAlgo, you will realise that sorting is a pre-processing step for many other advanced algorithms for harder problems, e.g. as the pre-processing step for Kruskal's algorithm, creatively used in Suffix Array data structure, etc.