EE2331 Data Structures and Algorithms

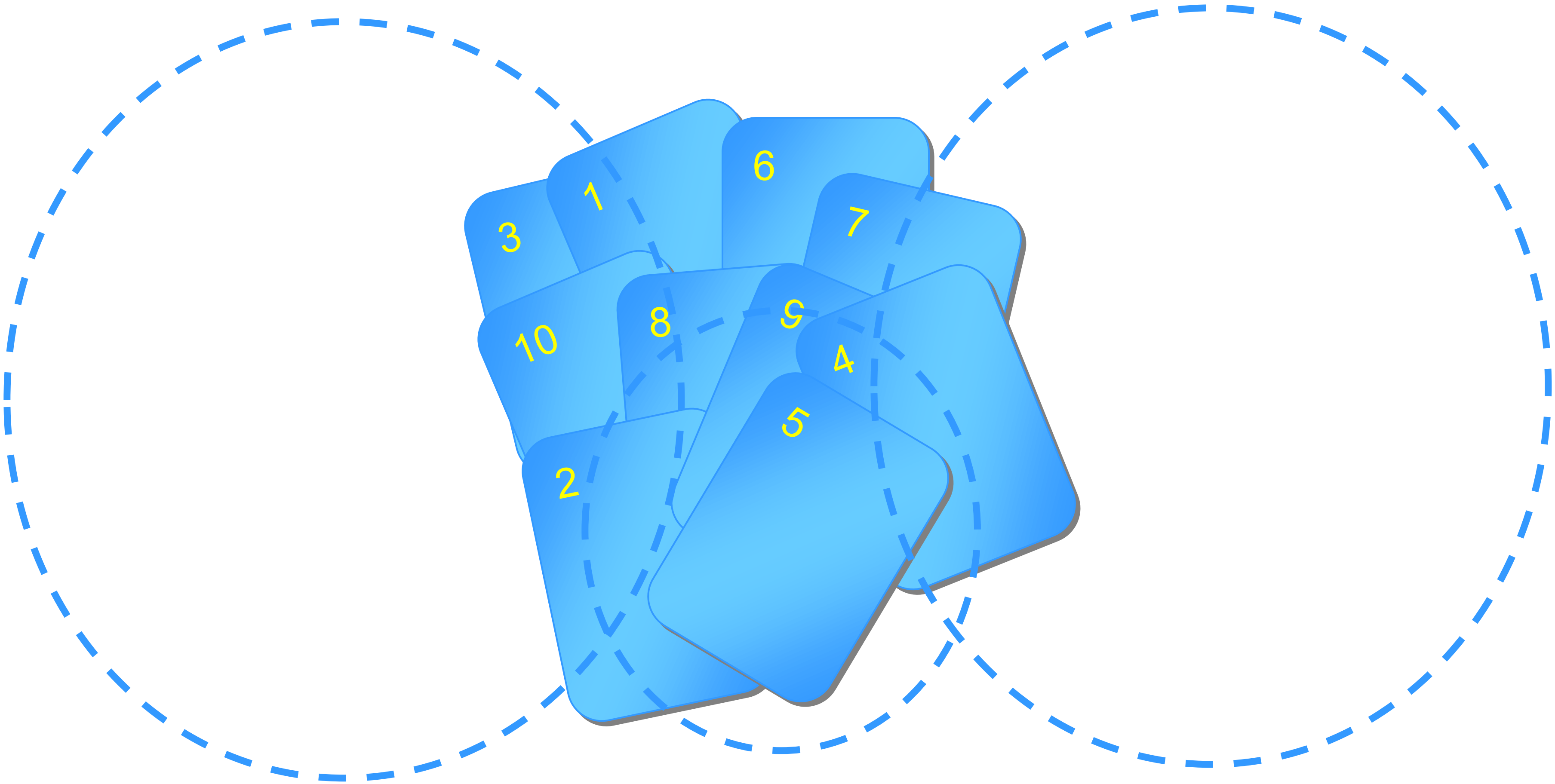
quick sort

Quicksort

The fastest known comparison-based sorting algorithm

Divide-and-conquer algorithm

Quicksort



Left <= pivot < Right

(recursion) (recursion)

Basic algorithm of quicksort

- Step 1: choose a pivot
- Step 2: partition the input array into low and high subarrays (divide)
- Step 3: recursively sort low and high subarrays (conquer)

Low: elements <= pivot

High: elements > pivot

Example of partition

Input array: {13, 81, 92, 43, 65, 31, 57, 26, 75, 0}

Assume pivot = 65. Partition this array using the pivot 65:

{13 0 43 31 26 5} 65 {92 75 81}

65 is in its correct position in a fully sorted array.

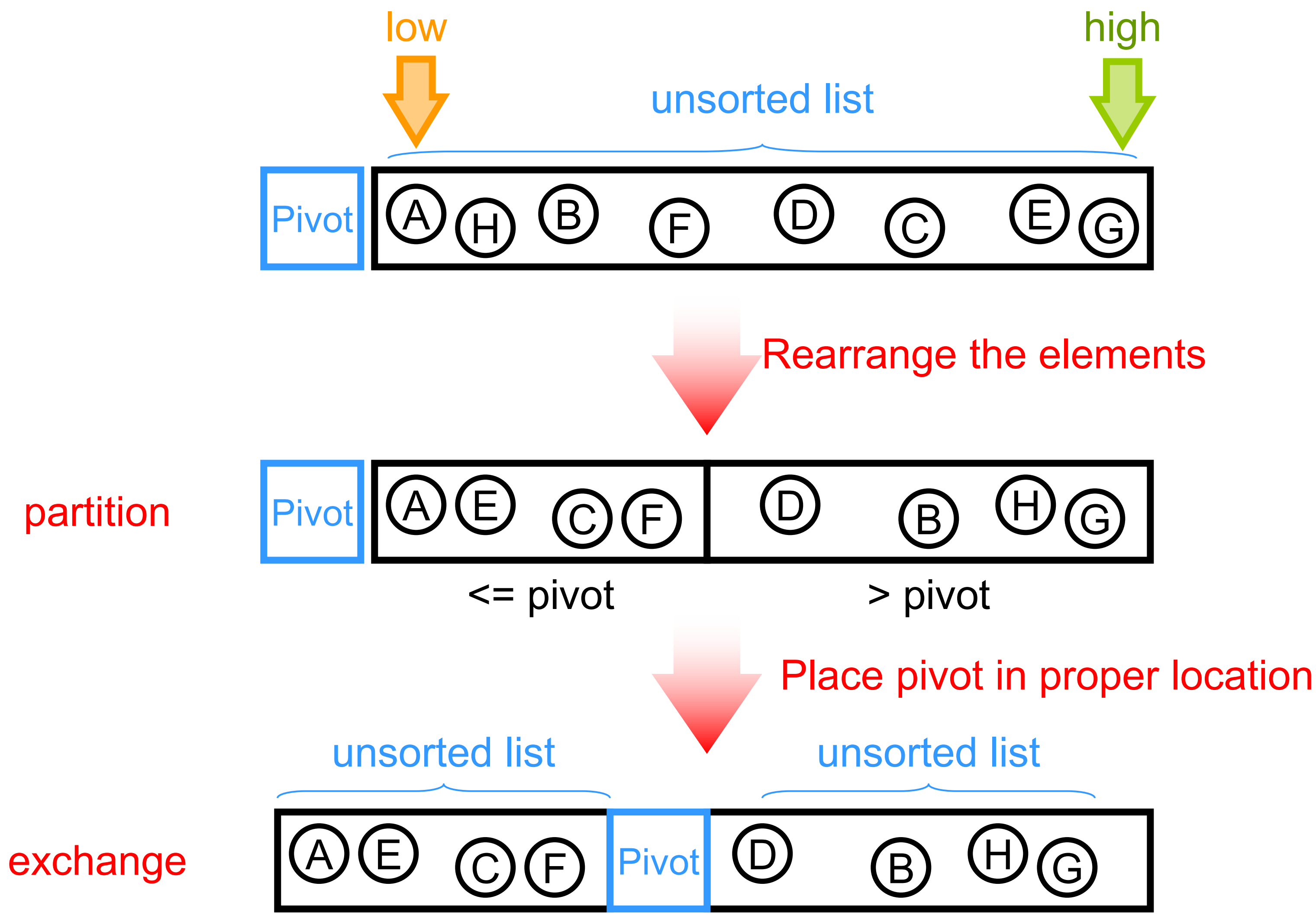
In-class exercise:
Provide a strategy (algorithm) for partition. Use pseudocode or English.

Input: an array (data[]) and a pivot (x).
Output: the array after partition

Exchange and Partition

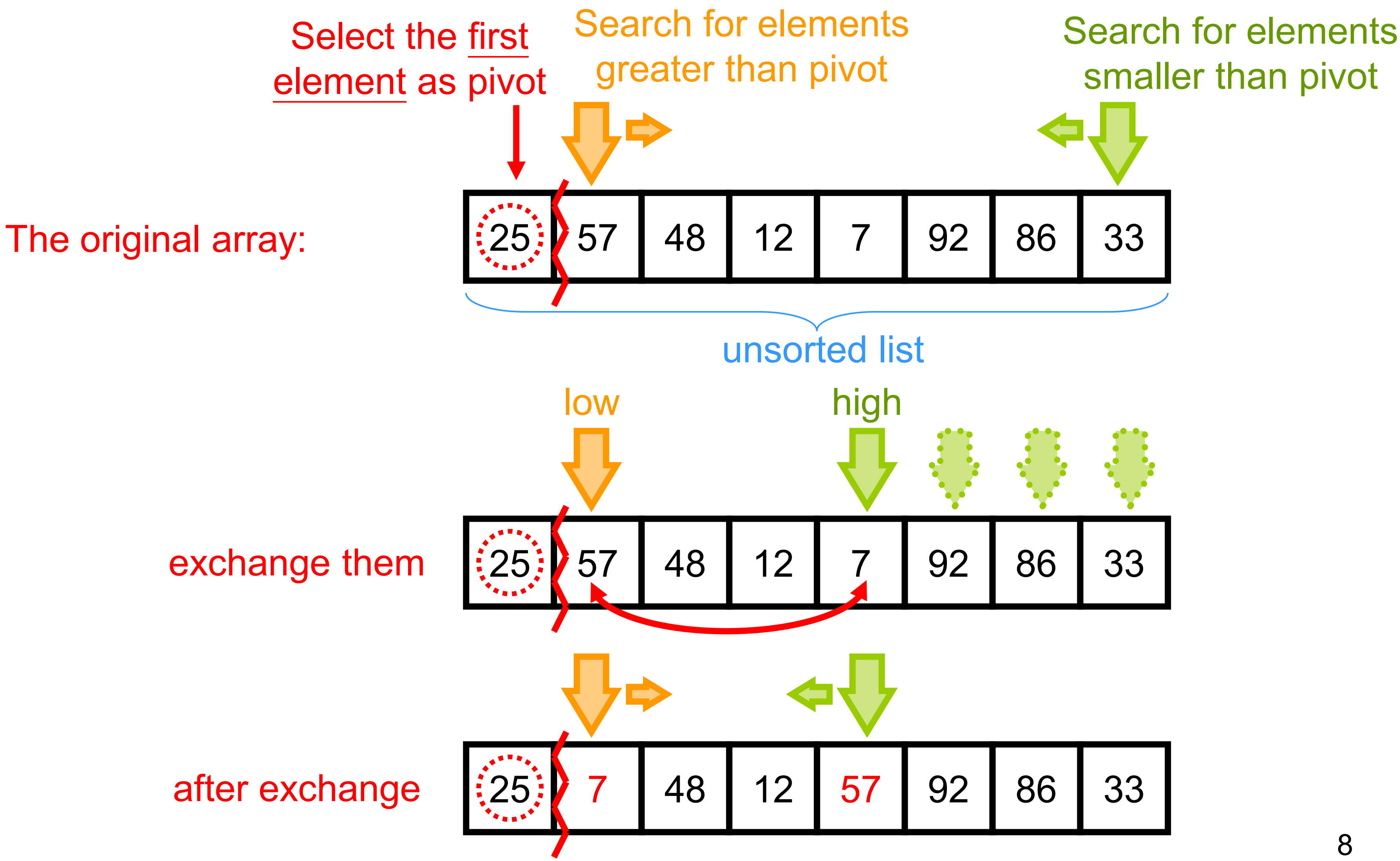
- A.K.A. ***partition-exchange sort***
  - Step 1) Exchange, then Step 2) Partition
- If the list has one or no elements (base case)
  - Do nothing (as already sorted)
- If the list has two or more elements (assume non-duplicate)
  - Pick an element as the **pivot**
  - Place the elements **smaller** than the pivot **before** it and the elements **larger** than the pivot **after** it (in any order) (by iteration)
  - Sort the sublist before the pivot (by recursion)
  - Sort the sublist after the pivot (by recursion)

The General Concept



A, *E*, *C*, *F* < =pivot *D*, *B*, *H*, *C* > pivot

Quicksort Example



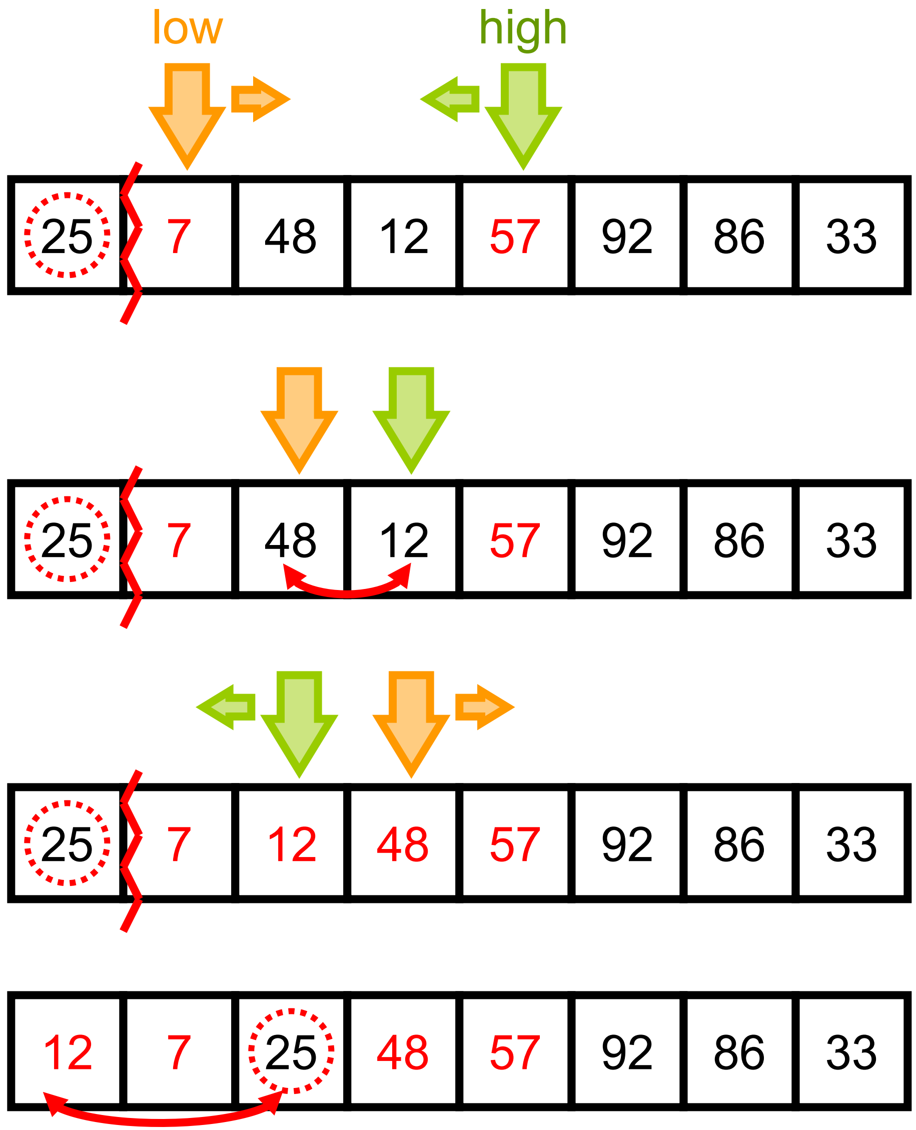
Quicksort Example

search again

exchange again

Search stops when
low ≥ high

exchange pivot
with high



Quicksort Example

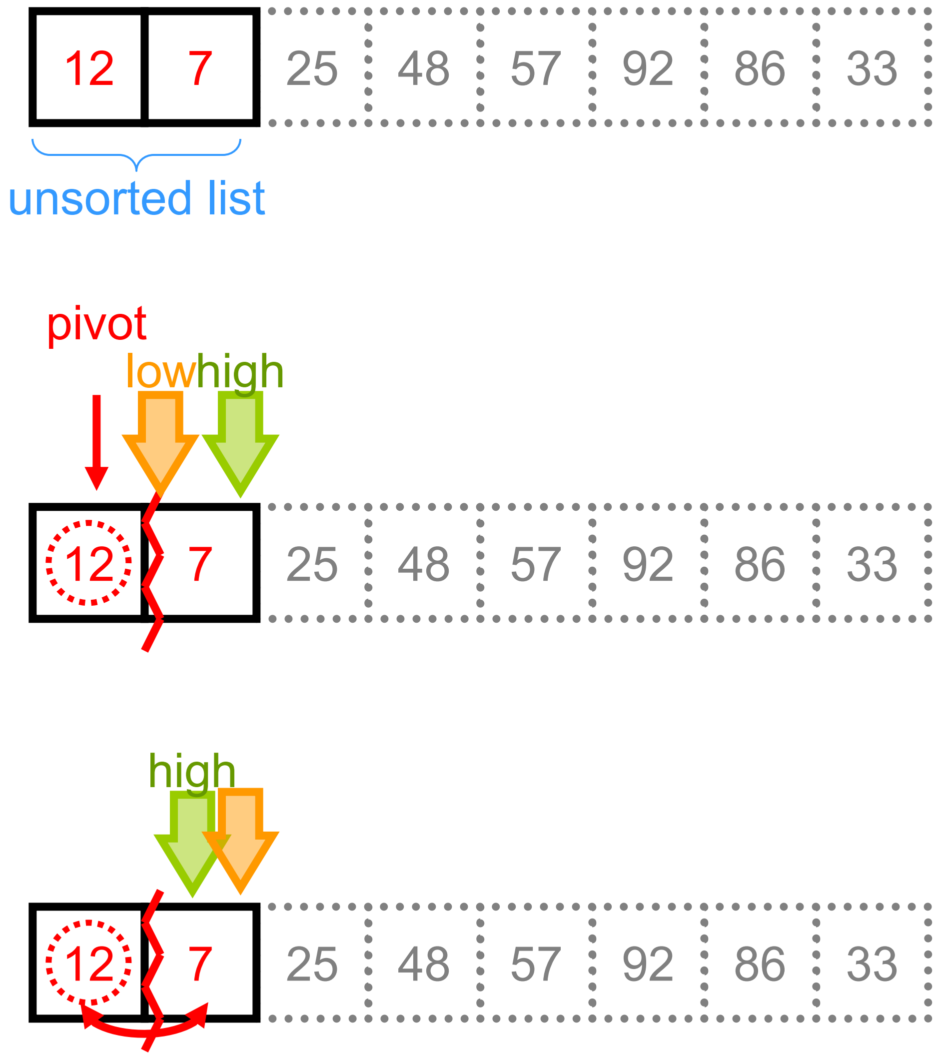
After 1st pass

$$\underbrace{\boxed{12}\,\boxed{7}}_{\text{unsorted list}}\ \underbrace{\boxed{25}}_{\text{sorted list}}\ \underbrace{\boxed{48}\,\boxed{57}\,\boxed{92}\,\boxed{86}\,\boxed{33}}_{\text{unsorted list}}$$

$$\underbrace{\{12, 7\}}_{\text{sort recursively}} < 25 \le \underbrace{\{48, 57, 92, 86, 33\}}_{\text{sort recursively}}$$

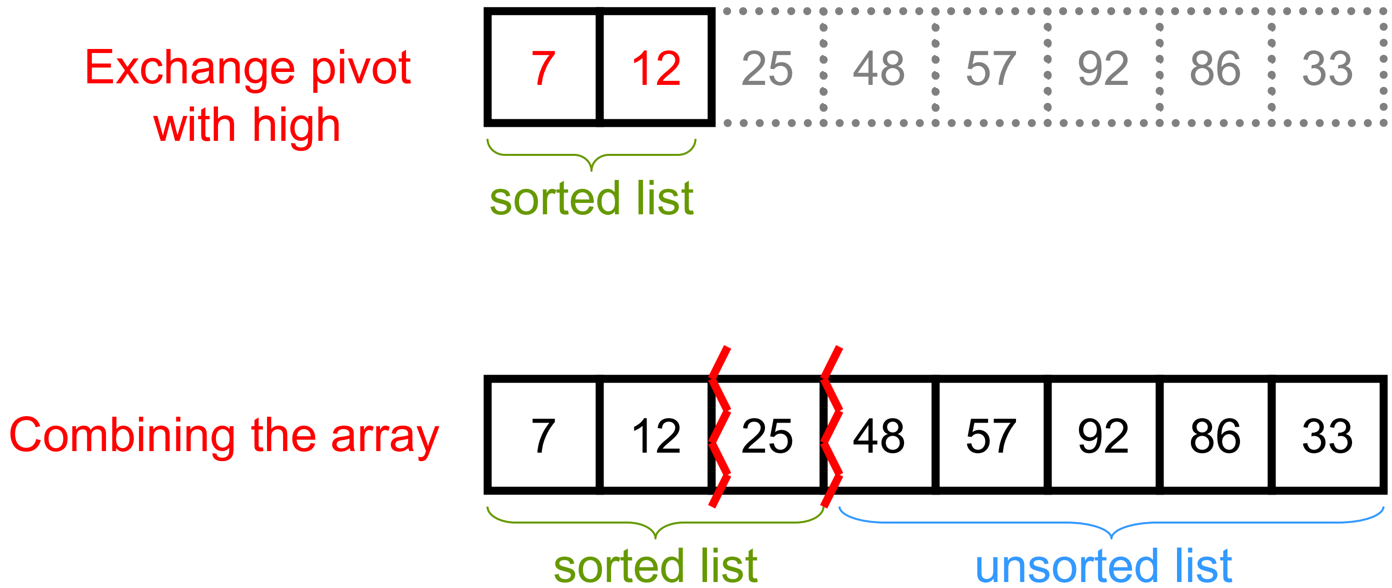
Sort the Left Sublist

Sort the left sublist



Because low == high, no searching is needed.

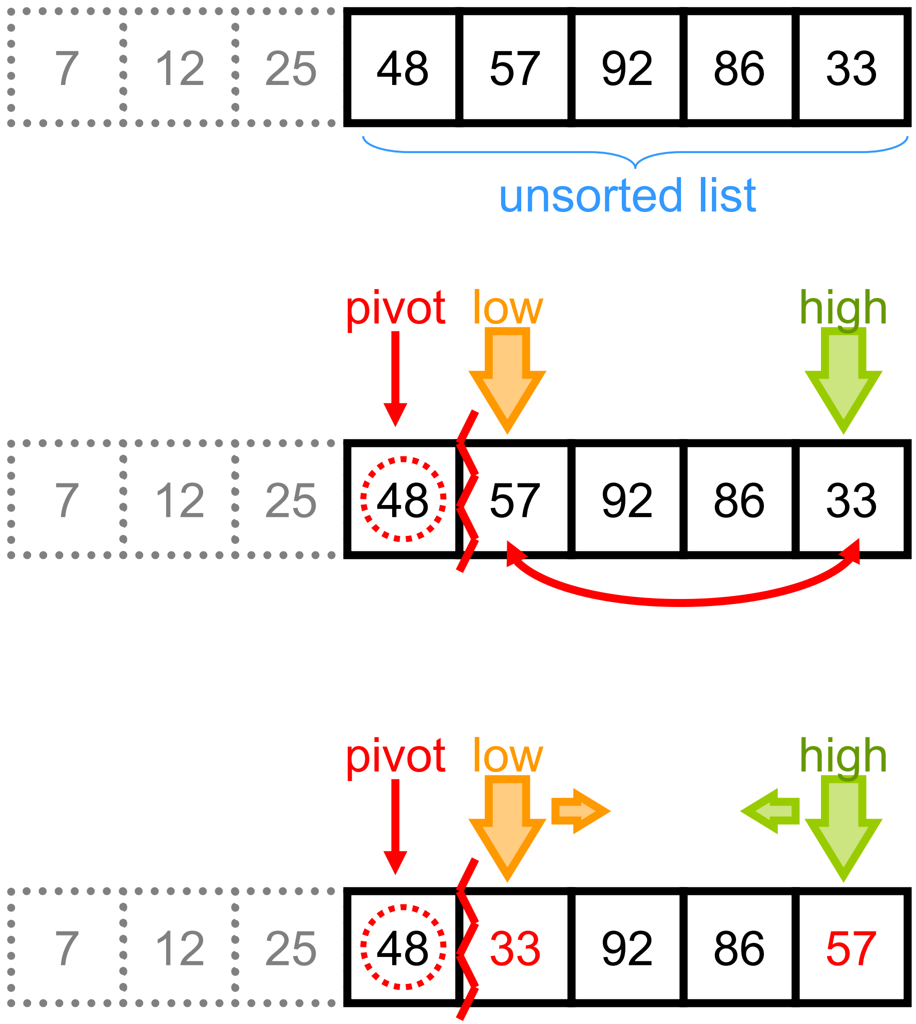
Sort the Left Sublist



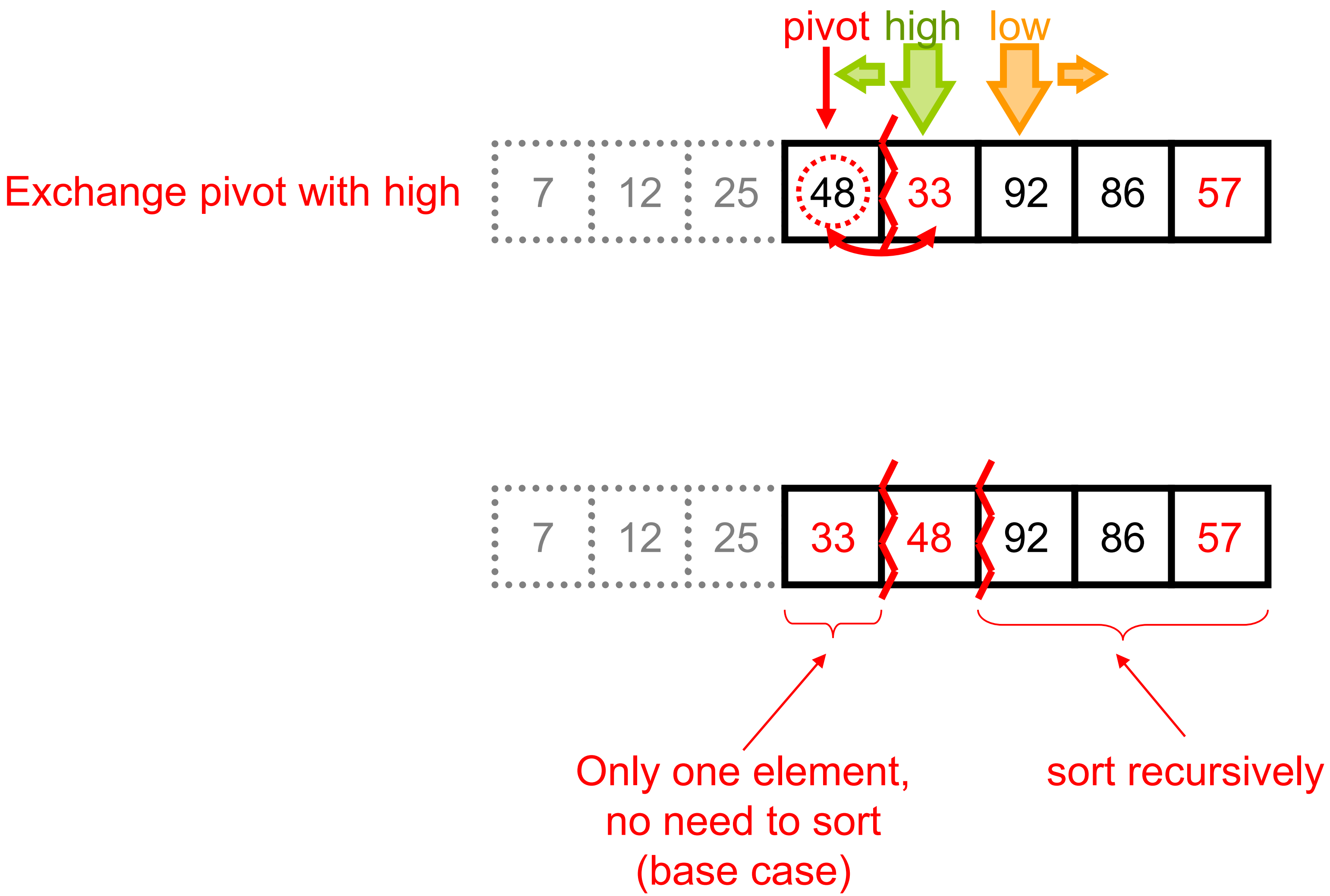
$$\{7, 12, 25\} \leq \underbrace{\{48, 57, 92, 86, 33\}}_{\text{sort recursively}}$$

Sort the Right Sublist

Search and exchange



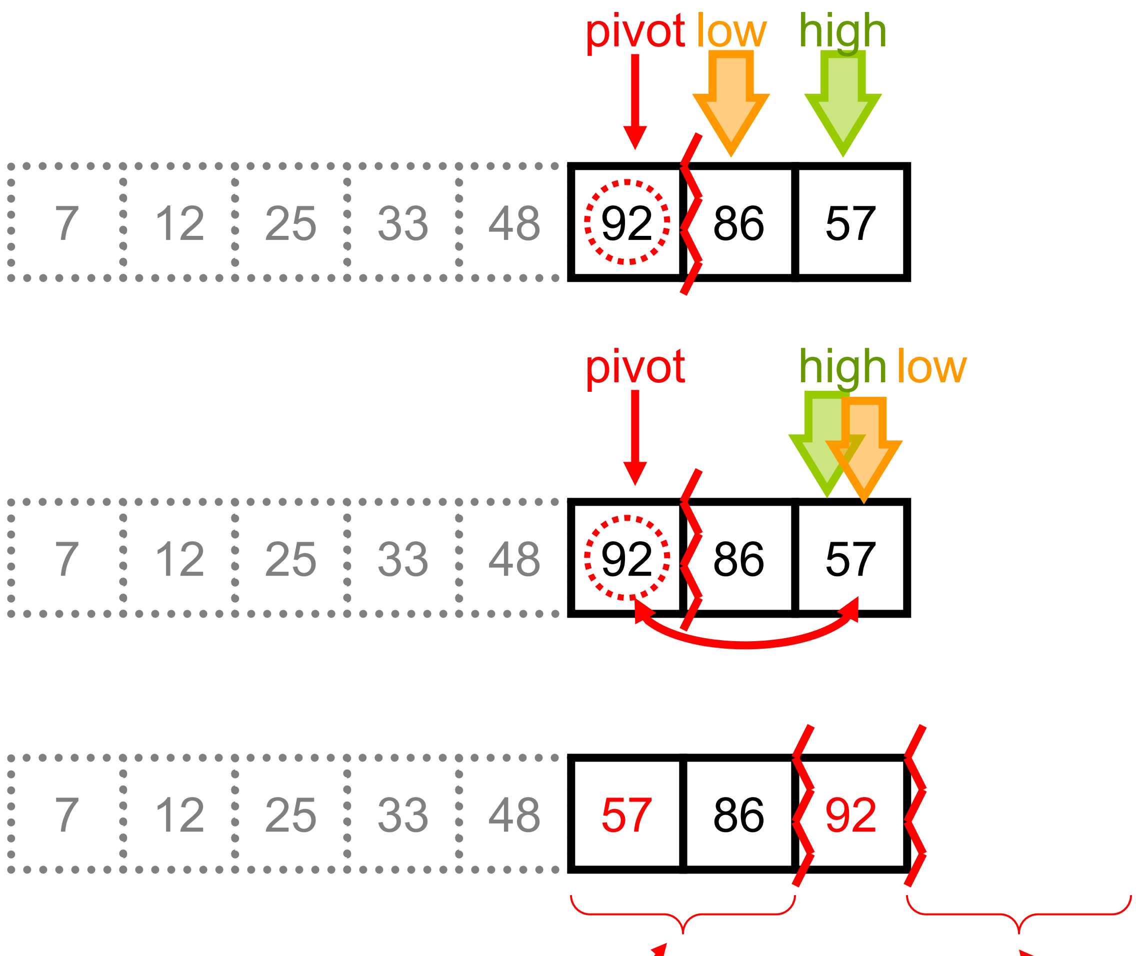
Sort the Right Sublist



Sort Another Right Sublist

Select the pivot, low and high before searching

Exchange pivot with high

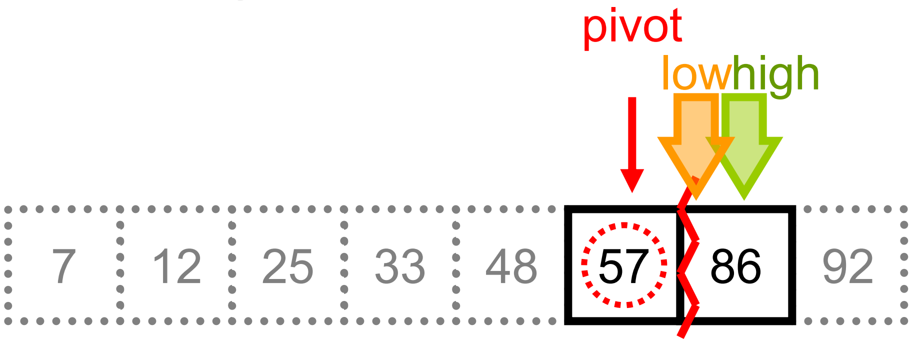


Sort this recursively

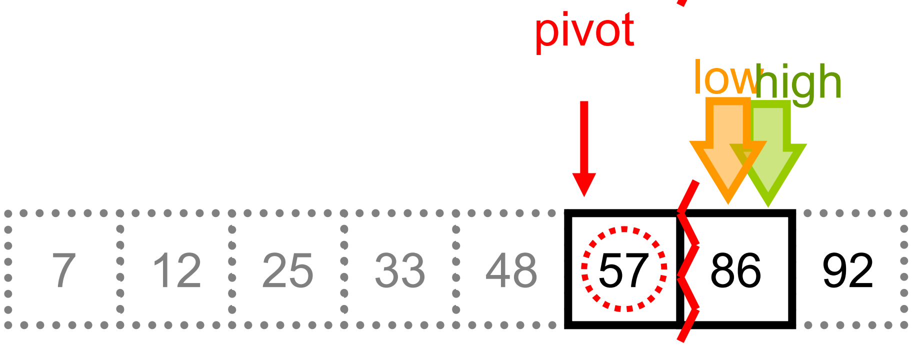
No need to sort (base case)

Sort the Last Sublist

Select the pivot, low and high before searching

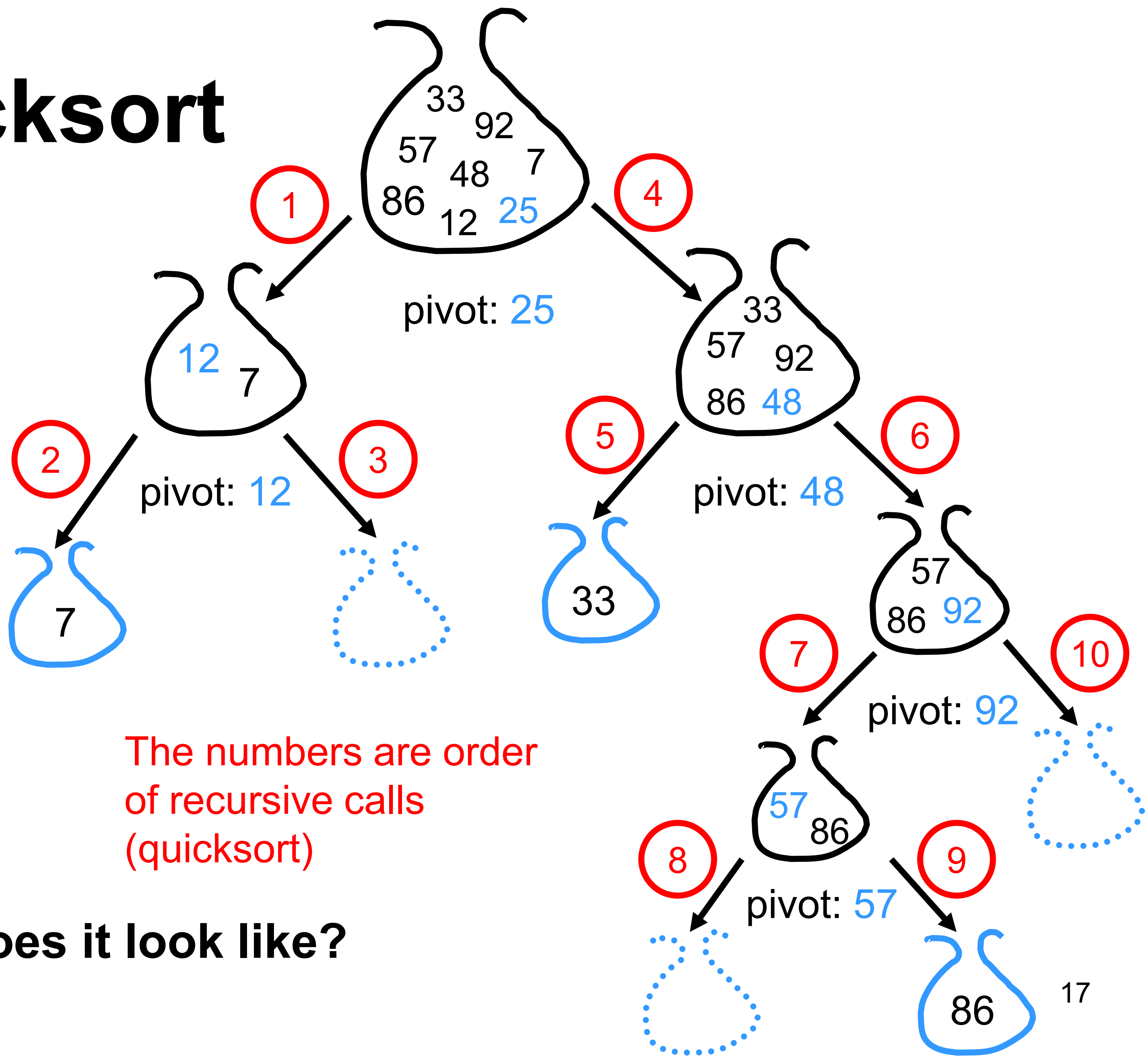


Low=high, no need to search. Data[pivot] < data[high], no need to swap either



Finally, the list is sorted correctly

Quicksort



The numbers are order of recursive calls (quicksort)

What does it look like?

Quicksort

- Divide-and-conquer sorting algorithm
- e.g. the unsorted array is data[*p*…*r*]
- Divide Stage
  - **Exchange** and **partition** the array data[] into **three** sub-arrays: data[*p*…*q*-1], data[*q*] and data[*q*+1…*r*] such that
  - All element in data[*p*…*q*-1] is less than or equal to data[q], and
  - All element in data[*q*+1…*r*] is greater than data[*q*]

Quicksort

- Conquer Stage
  - The two sub-arrays data[*p*…*q*-1] and data[*q*+1…*r*] are sorted recursively
- Combine Stage
  - The sub-arrays are sorted in place
  - No extra memory needed (except swapping)
  - No work is need to combine them

The Procedure

```
void quicksort(int data[], int p, int r) {  // p: start, r: end index
    int pivot, low, high, q;

    if (p >= r) return;  //base case

    pivot = p;              //set first element as pivot
    low = p + 1;
    high = r;

    while (low < high) {
        while(data[low] <= data[pivot] && low < r) low++;
        while(data[high] > data[pivot] && high > p) high--;
        if (low < high) swap(data[low], data[high]);
    }
    if (data[pivot] > data[high]) //swap pivot with high
            swap(data[pivot], data[high]);

    q = high;
    quicksort(data, p, q-1);
    quicksort(data, q+1, r);
}
```

divide (exchange & partition) (iteration)

conquer (recursion)

In-class exercise:
Apply the quicksort algorithm to array {13, 26, 31, 43, 57}.
Show the content of the array in each step.

```
void quicksort(int data[], int p, int r) {  // p: start, r: end index
    int pivot, low, high, q;

    if (p >= r) return;  //base case

    pivot = p;              //set first element as pivot
    low = p + 1;
    high = r;

    while (low < high) {
        while(data[low] <= data[pivot] && low < r) low++;
        while(data[high] > data[pivot] && high > p) high--;
        if (low < high) swap(data[low], data[high]);
    }
    if (data[pivot] > data[high]) //swap pivot with high
        swap(data[pivot], data[high]);

    q = high;
    quicksort(data, p, q-1);
    quicksort(data, q+1, r);
}
```

A “visualization” of quicksort

https://www.youtube.com/watch?v=3San3uKKHgg

Question: is the above the same as our quicksort?

Running time analysis

```
T(n)        void quicksort(int data[], int p, int r) {  // p: start, r: end index
                int pivot, low, high, q;

1               if (p >= r) return;  //base case

                pivot = p;              //set first element as pivot
                low = p + 1;
                high = r;

n               while (low < high) {
                    while(data[low] <= data[pivot] && low < r) low++;
                    while(data[high] > data[pivot] && high > p) high--;
                    if (low < high) swap(data[low], data[high]);
                }
                if (data[pivot] > data[high]) //swap pivot with high
                        swap(data[pivot], data[high]);

                q = high;
T(|low|)        quicksort(data, p, q-1);
T(|high|)       quicksort(data, q+1, r);
            }
```

Running time analysis

$$T(n)=\begin{cases} 1, & n = 0 \ or \ 1 \\ 1 + n + T(|low|) + T(|high|), & otherwise \end{cases}$$

- Best case analysis:

$$|low| = |high| = \frac{n}{2}$$

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

Refer to the merge sort analysis:

$T(n) = O(n log_2 n)$

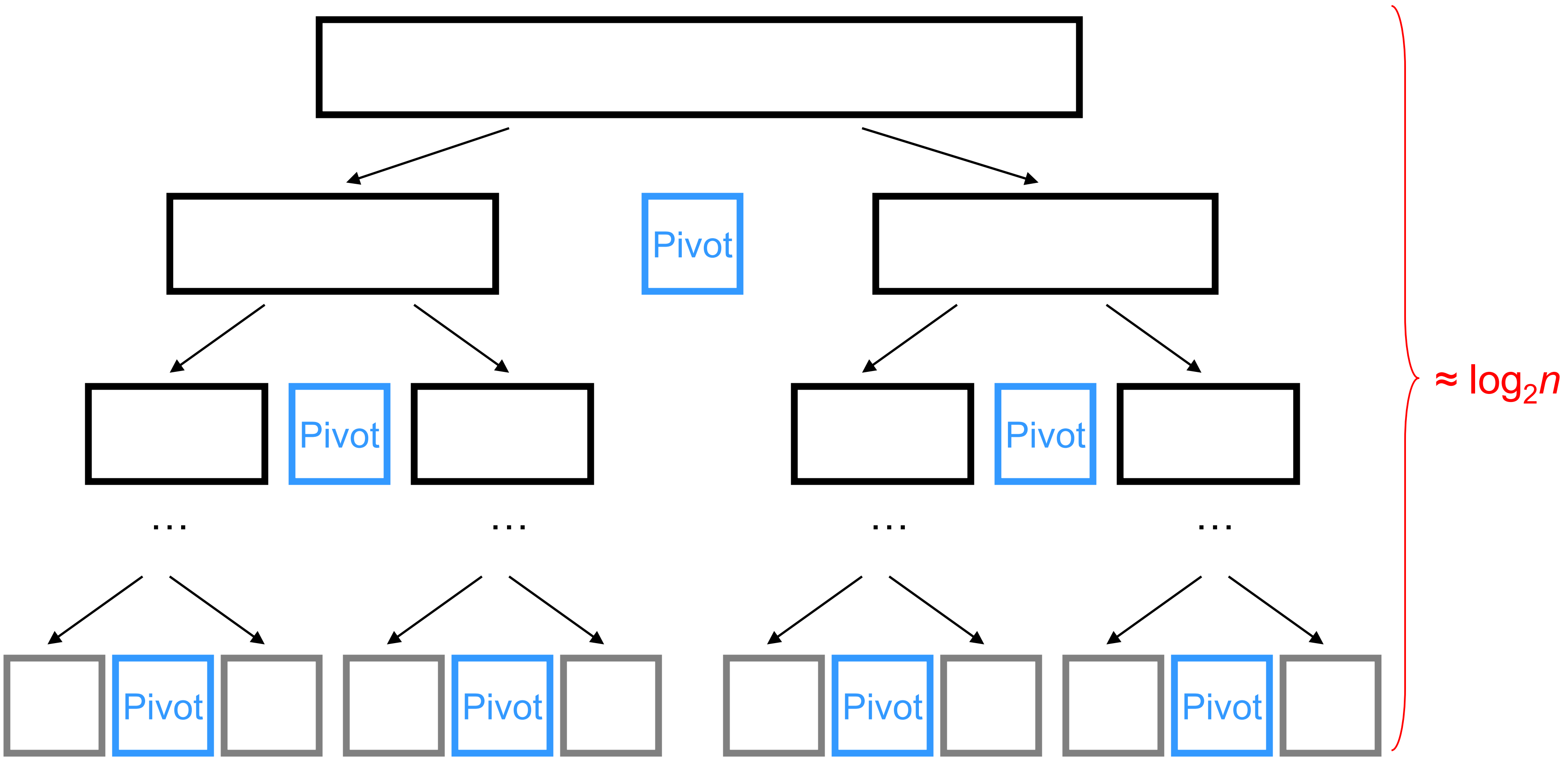
Running time analysis

- Worst case analysis:

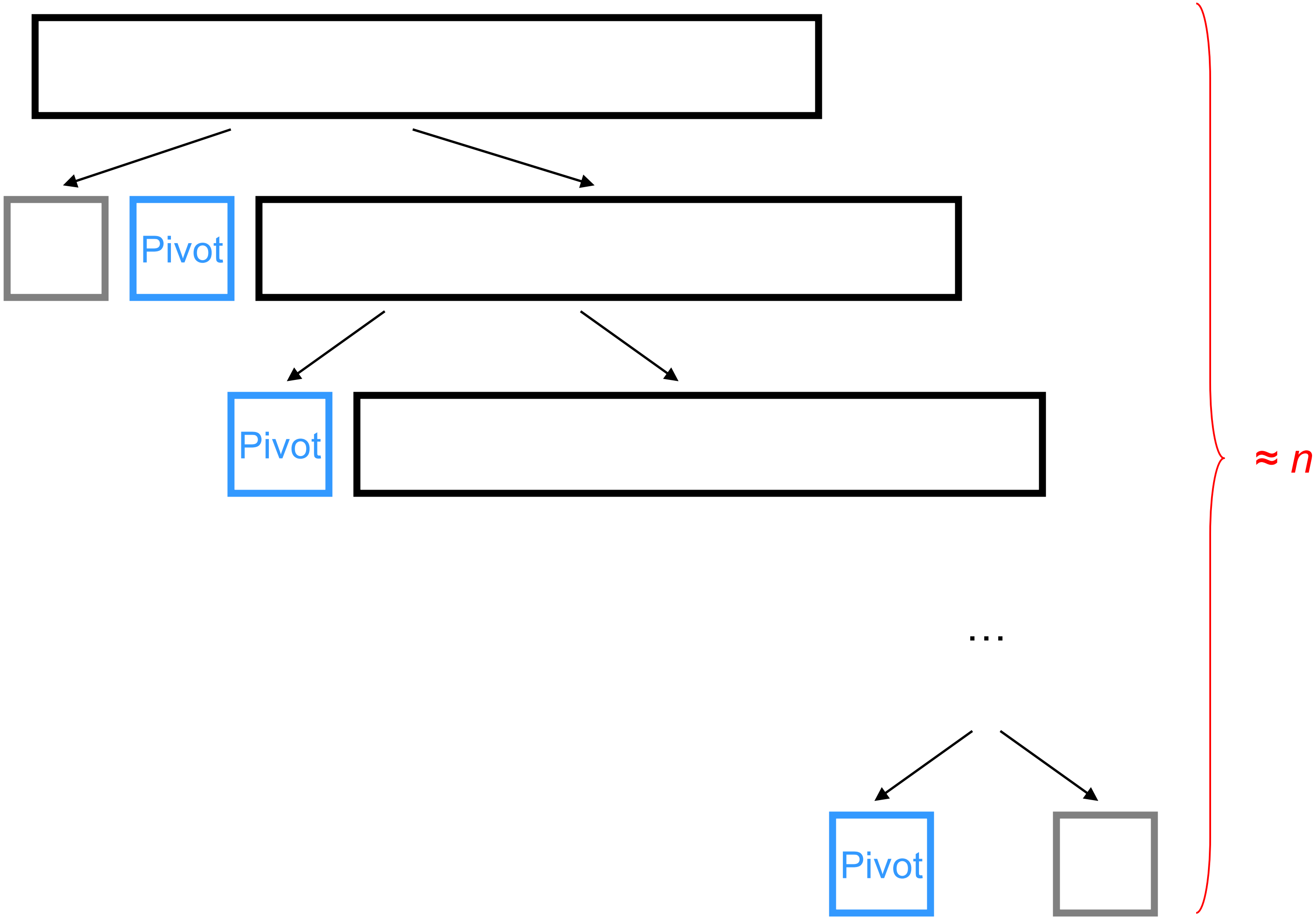
$$|low| = n - 1 \text{ or } |high| = n - 1$$

$$\begin{aligned} T(n) &= 1 + n + T(|low|) + T(|high|) \\ &= 1 + n + T(n-1) + T(0) \\ &= T(n-1) + n + 2 \\ &= T(n-2) + n + 1 + n + 2 \\ &= \dots \dots \\ &= O(n^2) \end{aligned}$$

A Good Pivot



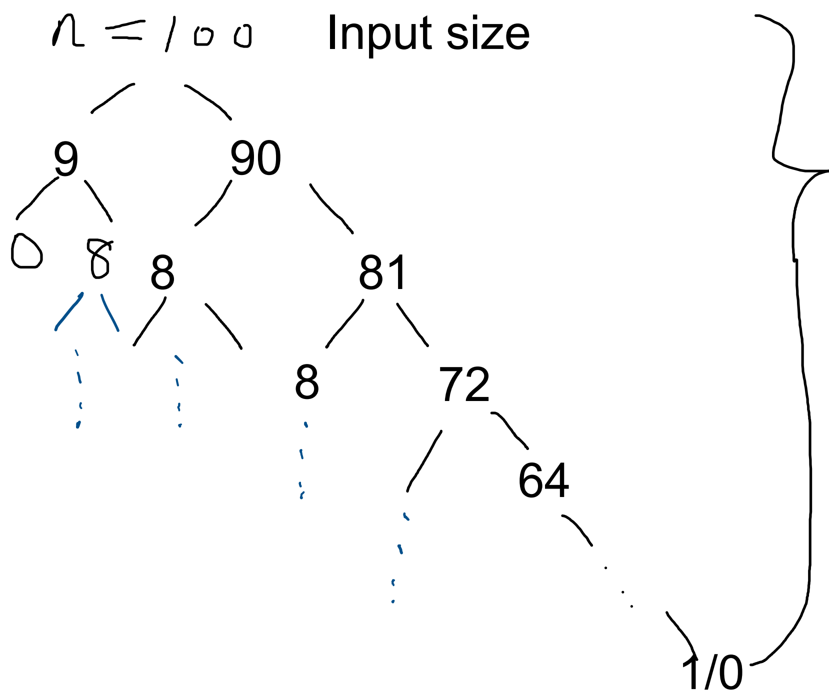
A Bad Pivot



Average (what if we are generally bad)

$T(n) = n+1 + T(\underline{9/10}\ n) + T(\underline{1/10\ n})$

$n = 100$ Input size



1 or 0 base case

n	$\left(\frac{9}{10}\right)^0 \cdot n$
$n*(9/10)^1$	$\left(\frac{9}{10}\right)^1 n$
$n*(9/10)^2$	$\left(\frac{9}{10}\right)^2 \cdot n$
$n*(9/10)^3$	$\left(\frac{9}{10}\right)^3 \cdot n$
.	
.	
.	
1	
$n*(9/10)^k = 1$?	$\underline{\left(\frac{9}{10}\right)^k \cdot n = 1}$
$k = \log_{10/9} n$	$k = \log_{\frac{10}{9}} n$

$T(n) = O(n \log_{10/9} n)$

$n \log_{10/9} n = O(n \log_2 n)$

$\left(\frac{9}{10}\right)^k = \frac{1}{n}$

$2^k = n \Rightarrow k = \log_2 n$

$\left(\frac{10}{9}\right)^k = n \Rightarrow k = \log_{\frac{10}{9}} n > \log_2 n$

$n\log_{10/9} n = O(\log_2 n)$, why? This can be proved following the definition of bigO notation (page 44, note 1). The proof is not very difficult but we won't require you to do this in this class.

In order to prove that $n\log_{10/9} n = O(\log_2 n)$, we need to find a constant C and n_0 so that when $n > n_0$, $C\, n \log_2 n >= n\log_{10/9} n$. By solving this formula, it is not hard to find out C. You can try to solve this simple formula based on basic properties of log functions. For example,

$\log_a b = \log_2 a/\log_2 b$

Complexity Analysis

- Partition
  - Low pointer moves to right, while high pointer moves to left
  - Total $n - 1$ comparisons
  - $O(n)$: linear time
- Exchange
  - Swapping nodes: O(1)
- How many passes in total?
  - The best case
    - Ideally, the two sub-lists will be of equal size if the median is chosen as pivot in each pass
    - There will be about $\log_2 n$ passes
    - So total time complexity is $O(n \cdot \log n)$
  - The worst case
    - If one of the sub-arrays is always empty, or has only one element
    - Total no. of passes is about n
    - Then quicksort takes $O(n^2)$ time

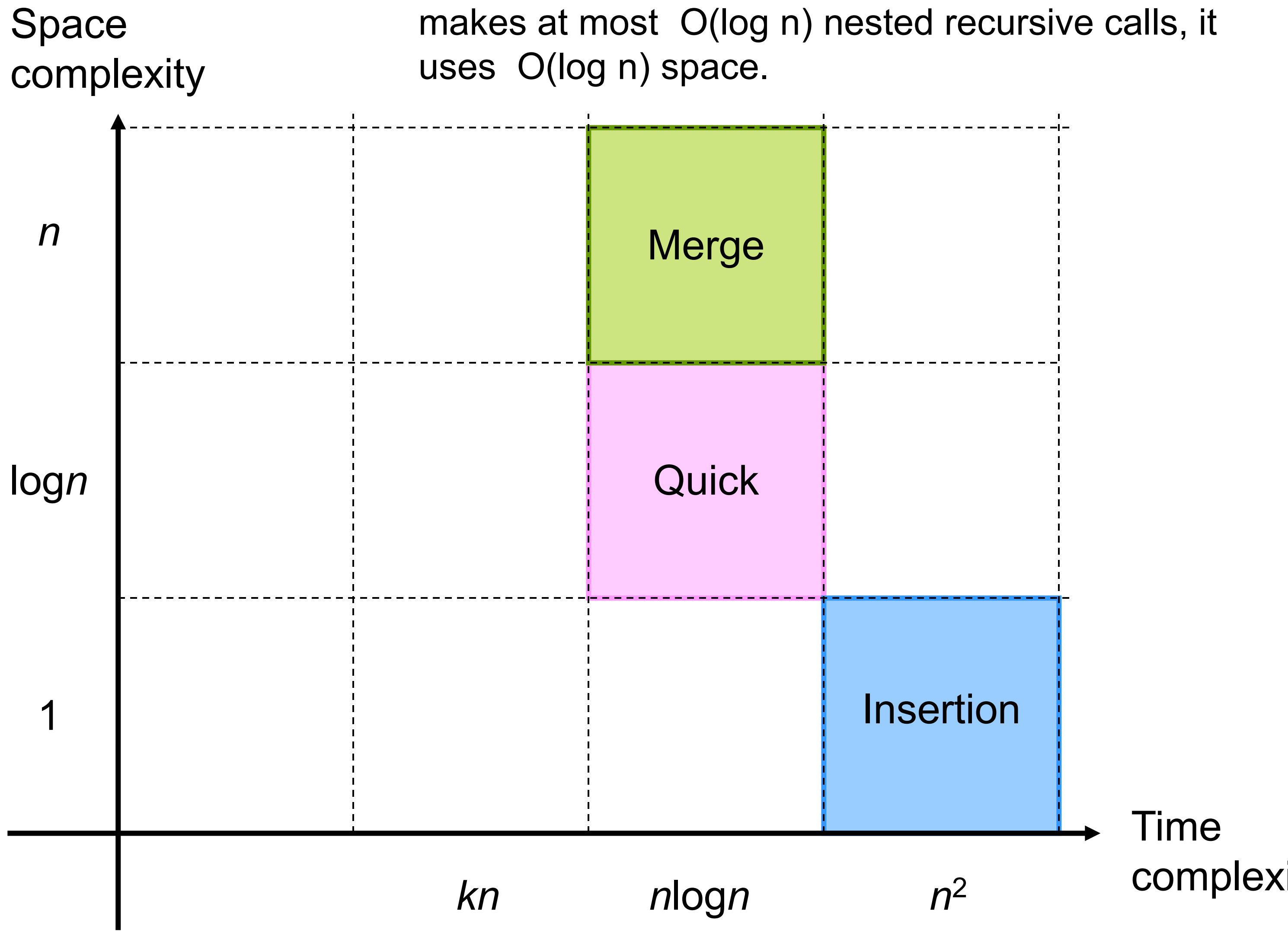
Choosing a Good Pivot

- By choosing the pivot carefully, we can obtain $O(n \cdot \log n)$ time in the average case
- The simplest (poor) version
  - Choose the first element as pivot
  - If the list is already sorted, the time complexity would be $O(n^2)$
- Two better versions
  - Choose the pivot randomly in each pass, or
  - Select the median between first, last and middle element as pivot
  - These two solutions cannot completely avoid the worst case
  - It can also be shown that the average cast complexity of quicksort is approximately equal to $1.38\ n \log_2 n$

- If the size of the array is large, quicksort is the fastest sorting method known today.

Summary

The version of quicksort with in-place partitioning uses only constant additional space before making any recursive call. However, if it has made O(log n) nested recursive calls, it needs to store a constant amount of information from each of them. Since the best case makes at most O(log n) nested recursive calls, it uses O(log n) space.



C++ Standard Template Library

- For short: STL: contains many general-purpose, templatized classes for commonly used data structures and algorithms
- We will focus on vector today. It is your job to learn “string” in STL.

- The vector is considered a *container* class: *hold other objects*.
  - A vector is a **dynamic array** designed to hold objects of any type
  - Can grow and shrink as needed, no need to use pointers to dynamically allocate memory

https://cplusplus.com/reference/vector/vector/

When and how to use a vector?

- **When to Use a Vector?**
- A C++ vector should be used under the following circumstances:
  - When dealing with data elements that change consistently.
  - If the size of the data is not known before beginning, the vector won't require you to set the maximum size of the container.
- In order to use vector, you need to include the following **header file**:
  - #include <vector>
- The vector is part of the std namespace, so you need to qualify the name. This can be accomplished as shown here:
  - **using namespace std**;
  - vector<int> myfirstVector;
  - Or
  - std::vector<int> myfirstVector;

Vector initialization

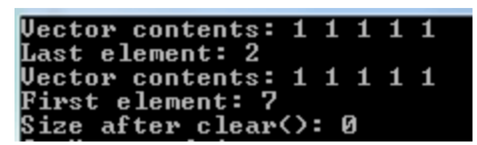
- vector <data-type> name (items)
  - E.g.
    - vector<int> first; //empty array/vector of integers
    - vector<float> second; //empty array of float
    - vector<int> third (4, 100) //four ints with value 100
    - vector<int> fourth (third) // a copy of ”third”
- Once you define a vector, you can access its elements like an array
  - E.g. third[0]=100
- Iterators
  - **vector:: begin():** it gives an iterator that points to the first element of the vector.
  - **vector:: end():** it gives an iterator that points to the past-the-end element of the vector.

Other commonly used functions

- **vector::push_back():** This modifier pushes the elements from the back.
- **vector::insert():** For inserting new items to a vector at a specified location.
- **vector::pop_back():** This modifier removes the vector elements from the back.
- **vector::erase():** It is used for removing a range of elements from the specified location.
- **vector::clear():** It removes all the vector elements.

Example

```
#include <iostream>
#include <vector>

using namespace std;
int main()
{
        vector<int> nums;

        nums.assign(5, 1);

        cout << "Vector contents: ";
        for (int a = 0; a < nums.size(); a++)
                cout << nums[a] << " ";

        nums.push_back(2);
        int n = nums.size();
        cout << "\nLast element: " << nums[n - 1];

        nums.pop_back();

        cout << "\nVector contents: ";
        for (int a = 0; a < nums.size(); a++)
                cout << nums[a] << " ";

        nums.insert(nums.begin(), 7);

        cout << "\nFirst element: " << nums[0];

        nums.clear();
        cout << "\nSize after clear(): " << nums.size();
}
```



String in STL

- https://cplusplus.com/reference/string/string/
- vector<string> stringArray
- You can compare strings the member function “string::compare”. See the example on the right
- “+”: concatenate strings

```
#include <iostream>
//source: https://www.geeksforgeeks.org/comparing-two-strings-cpp/

using namespace std;

void compareFunction(string s1, string s2)
{
        // comparing both using inbuilt function
        int x = s1.compare(s2);

        if (x != 0) {
                cout << s1
                        << " is not equal to "
                        << s2 << endl;
                if (x > 0)
                        cout << s1
                        << " is greater than "
                        << s2 << endl;
                else
                        cout << s2
                        << " is greater than "
                        << s1 << endl;
        }
        else
                cout << s1 << " is equal to " << s2 << endl;
}

// Driver Code
int main()
{
        string s1("ABC");
        string s2("BCA");
        compareFunction(s1, s2);
        string s3("Sam");
        string s4("Sam");
        compareFunction(s3, s4);
        return 0;
}
```

Example of string:+=

```cpp
// string::operator+=
#include <iostream>
#include <string>

int main ()
{
  std::string name ("John");
  std::string family ("Smith");
  name += " K. ";         // c-string
  name += family;         // string
  name += '\n';           // character

  std::cout << name;
  return 0;
}
```

Output:

John K. Smith

A note about STL vector

- Relevant to the swap function in quicksort

```
// C++ program to demonstrate that when vectors
// are passed to functions without &, a copy is
// created.
#include<vector>
using namespace std;

// The vect here is a copy of vect in main()
void func(vector<int> vect)
{
   vect.push_back(30);
}

int main()
{
    vector<int> vect;
    vect.push_back(10);
    vect.push_back(20);

    func(vect);

    // vect remains unchanged after function
    // call
    for (int i=0; i<vect.size(); i++)
       cout << vect[i] << " ";

    return 0;
}
```

```
// C++ program to demonstrate how vectors
// can be passed by reference.
#include<vector>
using namespace std;

// The vect is passed by reference and changes
// made here reflect in main()
void func(vector<int> &vect)
{
   vect.push_back(30);
}

int main()
{
    vector<int> vect;
    vect.push_back(10);
    vect.push_back(20);

    func(vect);

    for (int i=0; i<vect.size(); i++)
       cout << vect[i] << " ";

    return 0;
}
```

Reference: https://www.geeksforgeeks.org/passing-vector-function-cpp/