Last Lecture

- We saw how to ...
  - Define the best, worst, and average case running time and space efficiency of
    - merge sort
Learning Outcomes

- At the end of the next few lectures, a student will be able to:
  - describe the behaviour of and implement simple sorting algorithms:
    - insertion sort
    - selection sort
  - describe the behaviour of and implement more efficient sorting algorithms:
    - quick sort
    - merge sort
  - analyze the best, worst, and average case running time (and space) of these sorting algorithms
Today’s menu

➡ Looking at
  ➡ Describing how quick sort function
  ➡ Analyzing the time efficiency of quick sort
  ➡ Discussing how to improve quick sort’s time efficiency
  ➡ Analyzing its space efficiency
  ➡ Discussing how to improve quick sort’s space efficiency
How Quick Sort works

- **Divide and conquer** algorithm
- Recursive in nature
- Array has \( n \) elements

1. **Pivot**: we select an element as the pivot
2. **Partitioning**: we partition the array around this pivot - by swapping elements such that elements < pivot are in one partition and elements > pivot are in the other

   Repeat partitioning each partition until the array is completely sorted
Many different versions of Quick Sort

- Different ways of selecting the pivot -> see later on in lecture
- Different ways of partitioning the array using the pivot
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi - 1 do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i

Lomuto partition scheme – thank you, wiki!
**Hoare partition scheme** – thank you, wiki!

**Algorithm quicksort**(A, lo, hi) is

if \( lo < hi \) then

\( p := \text{partition}(A, lo, hi) \)

quicksort(A, lo, p)

quicksort(A, p + 1, hi)

**Algorithm partition**(A, lo, hi) is

pivot := A[(lo + hi) / 2]

i := lo - 1

j := hi + 1

loop forever

\( \text{do} \)

\( i := i + 1 \)

while A[i] < pivot

\( \text{do} \)

\( j := j - 1 \)

while A[j] > pivot

if i >= j then

return j

swap A[i] with A[j]
Let’s have a look at Quick Sort
Quick Sort – Partitioning step

- **Problem statement** -> sort this array:

- So we partition this array this way: (where $P_1$ is pivot)

- Then the problem becomes -> sort these 2 smaller arrays:

- So we partition these 2 smaller arrays this way:

- Then the problem becomes -> sort these even smaller arrays:

- So we continue solving sort problems by partitioning smaller and smaller arrays until each sort problem is solved, i.e., until each partition contains only 1 element
Quick Sort – Recursing up

- Once we have solved these small sort problems:
  \[ p_2 \ p_2 \ p_5 \ p_5 \ p_1 \ p_6 \ p_6 \ p_3 \ p_3 \]

- Then we have automatically solved these larger sort problems:
  \[ p_2 \ p_2 \ p_2 \ p_3 \ p_3 \ p_3 \]

- And we have automatically solved these even larger sort problems:
  \[ p_1 \ p_1 \ p_1 \]

- And finally, we have automatically solved the original sort problem, i.e., the one stated in the problem statement:
  \[ \]

\[
\begin{array}{ccccccccc}
< p_2 & p_2 & > p_5 & p_5 & p_1 & < p_6 & p_6 & p_3 & > p_3 \\
< p_2 & p_2 & > p_2 & < p_3 & p_3 & > p_3 \\
< p_1 & p_1 & > p_1 \\
\end{array}
\]
Each time we partition ... 
- We divide in half
  - The partition with elements < pivot is about the same size as the partition with elements > pivot
  - Pivot is (more or less) in the middle
- ~ n comparisons are made at each level

How many times do we partition?
- We divide in half until the partitions have size 1
- How many times does n have to be divided in half before the result is 1?
- Answer: $\log_2(n)$ times

Quick sort performs around $n \times \log_2(n)$ operations in the best case scenario
Details of time efficiency analysis: how many times do we partition?

Before we divide:

Initially:
\[ N = \frac{N}{2^0} \]

After 1st division:
\[ \frac{N}{2} = \frac{N}{2^1} \]

After 2nd division:
\[ \frac{N}{4} = \frac{N}{2^2} \]

...  

After 14th division:
\[ \frac{N}{2^{14}} = 1 \]

\[ \downarrow \]

\[ N = 2^T \]

Answer: \( T \) steps

Express \( T \) as a function of \( N \):
\[ \log_2 N = \log_2 2^T \]

\[ T = \log_2 N \]
Quick Sort - Worst Case Scenario

...
Each time we partition …
- We don’t divide in half
  - Either the partition with elements < pivot is empty or the partition with elements > pivot is empty
  - Pivot, when swapped, lands at either end of the partition
- ~ n comparisons are made at each level

How many times do we partition?
- Considering that at each partition, all we do is remove the pivot from the partition i.e., n is decremented by 1
- How many times does n have to be decremented by 1 before the result is 1?
  - Answer: n - 1 times

Quick sort performs around \( n \times (n-1) \) operations -> \( n^2 \) operations in the worst case scenario
Time Efficiency Analysis of Quick Sort – Average Case Scenario

- Average case scenario of quick sort is closer to its best case scenario than to its worst case scenario
Time Efficiency Analysis of Quick Sort

- Time efficiency of
  - Best case:
  - Average case:
  - Worst case:
Improving Quick Sort – Choice of Pivot

- Different ways of selecting the pivot:
  - **Way 1 (used so far):** array[0] or array[size-1]
  - **Way 2:** use largest of 1st two elements

- Advantage of ways 1 and 2:

- Disadvantage of ways 1 and 2:
Improving Quick Sort – Choice of Pivot

- **Way 3:** choose three elements
  - First, last and middle elements of array
  - Sort order them
  - Pick their median as pivot
  - $O(1)$

- **Advantage:**

- **Disadvantage:**

To be used if problem statement leads us to believe that the array may be partially sorted
Space Efficiency Analysis of Quick Sort

- Quick sort is in-place sort algorithm
- Quick sort is often implemented as a recursive function
- How much space (memory) does quick sort require to execute?
- Therefore, its space efficiency is ->
Improving Quick Sort – Space efficiency

- To improve quick sort, we could reduce the number of recursive calls
  - Improvement 1: median - of - 3
  - Improvement 2:
    - Stop using quick sort when size of partition small (i.e. < 10)
    - Use insertion sort for these small partitions
  - Improvement 3:
    - Quick sort calls itself twice
    - Second call:
      - Recursive call done as the last operation -> tail recursion
      - Replace tail recursion with a loop
  - Improvement 4:
    - First call: call quick sort recursively using the smaller partition
Is quick sort stable?
√ Learning Check

- We can now ...
  - Describe how quick sort function
  - Analyze the time efficiency of quick sort
  - Discuss how to improve quick sort’s time efficiency
  - Analyze its space efficiency
  - Discuss how to improve quick sort’s space efficiency
Next Lectures

- Let’s have a look at another way of organizing our data
- Another category of data organization
Extra Material
Example of Improvement 1 and 2 in Java

```java
public void recQuickSort(int left, int right)
{
    int size = right-left+1;

    if (size < 10) // Insertion Sort if small
        insertionSort(left, right);
    else // Quick Sort if large
    {
        double median = medianOf3(left, right);
        int partition =
            partitionIt(left, right, median);
        recQuickSort(left, partition-1);
        recQuickSort(partition+1, right);
    }
} // end recQuickSort()
```
Example of Improvement 3

```c
/* The main function that implements QuickSort
 arr[] --> Array to be sorted,
   low --> Starting index,
   high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    while (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);

        low = pi+1;
    }
}

// For complete running code, see
http://code.geeksforgeeks.org/qrlM31
```
Side Trip -> Tail Recursion

- Recursive method in which ...
  - there is no computation to be done once the recursive call has returned, and
  - the entire state of computation is represented in parameter(s) of method
Example of Non-Tail Recursive Factorial Method in Java

```java
private static int fac( int n ) {
    // Base case
    if ( ( n == 0 ) || ( n == 1 ) )
        return 1;

    // Recursive case
    return ( n * fac( n - 1 ) );
}
```

Call: fac( n );
Example of Tail Recursive Factorial Method in Java

```java
private static int fac( int n, int result ) {
    // Base case
    if ( ( n == 0 ) || ( n == 1 ) )
        return result;

    // Recursive case
    return fac( n - 1, n * result );
}

Call: fac( n, 1 );
```
Advantages of Tail Recursion - 1

- Can be transformed into equivalent iterative method by replacing recursive method call using a loop
Transforming Tail Recursive Method into Iterative Method

1. Transform "the answer" (here result) into a local variable in the calling method (wrapper) and initialize it to the value in the base case
   
   ```java
   int result = 1;
   ```

2. Replace the recursive call with a loop which builds the answer:
   
   ```java
   for ( ; n > 1; n-- ) {
       // compute factorial
       result = result * n;
   }
   ```

3. Return the answer
   
   ```java
   return result;
   ```
Example of Iterative Factorial Method

in Java

// PRE condition: n must be non-negative
public static int fac( int n )
{
    int result = 1;

    // Ensure PRE condition is satisfied
    for ( ; n > 1; n-- ) {
        // compute factorial
        result = result * n;
    }

    return result;
}
Advantages of Tail Recursion - 2

Tail Recursion Optimization

- A compiler may recognize “tail recursion” and optimize the code by replacing the calling statement with the called method (with some modifications), so that instead of nesting the stack deeper, the current stack frame is reused when the code execute, i.e., no other stack frame than the first one is needed.

- This allows a software developer to write (tail) recursive methods without worrying about space inefficiency during execution.
  - The tail recursive code is then as efficient as its iterative version.

Example of Improvement 4

/* The main function that implements QuickSort */
int arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    while (low < high)
    {
        /* pi is partitioning index, arr[p] is now at right place */
        int pi = partition(arr, low, high);

        if (pi - low < high - pi)
        {
            quickSort(arr, low, pi - 1);
            low = pi + 1;
        }
        else
        {
            quickSort(arr, pi + 1, high);
            high = pi - 1;
        }
    }
}

// For complete running code, see
http://code.geeksforgeeks.org/LHxwPk