Merge Sort

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 2 & 4 & 6 \\
3 & 5 & 7 & 8 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
2 & 4 & 1 & 6 & 5 & 8 & 3 & 7 \\
2 & 4 & 1 & 6 & 8 & 5 & 3 & 7 \\
\end{array}
\]

**MergeSort(A, l, r)**

{  
    if(l<r)
    {
        int m=(l+r)/2;
        MergeSort(A,l,m);
        MergeSort(A,m+1,r);
        Merge(A,l,m,r);
    }
}

Worst Case Running Time: \(O(n \log n)\)
But not in-place space complexity \(O(n)\)
Let’s assume we have two sorted half arrays. Then, let’s merge the L and R into A.

```
Merge(A, l, m, r)
{
    int i, j, k;
    int nL=m-l+1;
    int nR=r-m;
    int L[nL], R[nR];

    for(i=0; i<nL; i++) {  
        L[i]=A[l+i];
    }

    for(j=0; j<nR; j++) {  
        R[j]=A[m+j+1];
    }

    i=0, j=0, k=l;
    while(i<nL && j<nR) {  
        if(L[i]<R[j]) {  
            A[k]=L[i];
            i++;
        } else {  
            A[k]=R[j];
            j++;
        }
        k++;
    }

    while(i<nL) {  
        A[k]=L[i];
        i++;
        k++;
    }

    while(j<nR) {  
        A[k]=R[j];
        j++;
        k++;
    }
}
```
The Recursion Pattern

- **Recursion**: when a method calls itself
- **Classic example**: the factorial function:
  \[ n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n \]
- **Recursive definition**:
  \[
  f(n) = \begin{cases} 
  1 & \text{if } n = 0 \\
  n \cdot f(n-1) & \text{else}
  \end{cases}
  \]
- **As a C++ method**:
  ```cpp
  // recursive factorial function
  int recursiveFactorial(int n) {
    if (n == 0) return 1; // basis case
    else return n * recursiveFactorial(n-1); // recursive case
  }
  ```
Content of a Recursive Method

- **Base case(s)**
  - Values of the input variables for which we perform no recursive calls are called **base cases** (there should be at least one base case).
  - Every possible chain of recursive calls **must** eventually reach a base case.

- **Recursive calls**
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.
Visualizing Recursion

- Recursion trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value

Example

```
recursiveFactorial (4)
  \|/
   | call
   v
recursiveFactorial (3)
  \|/
   | call
   v
recursiveFactorial (2)
  \|/
   | call
   v
recursiveFactorial (1)
  \|/
   | call
   v
recursiveFactorial (0)
```

- return 4*6 = 24 → final answer
- return 3*2 = 6
- return 2*1 = 2
- return 1*1 = 1
- return 1
Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.

- Recur once
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.
Another Linear Recursion

**Algorithm** \texttt{LinearSum}(A, n):

**Input:**
- A integer array \( A \) and an integer \( n \geq 1 \), such that \( A \) has at least \( n \) elements

**Output:**
- The sum of the first \( n \) integers in \( A \)

\[ \text{if } n = 1 \text{ then} \]
\[ \text{return } A[0] \]

\[ \text{else} \]
\[ \text{return } \texttt{LinearSum}(A, n - 1) + A[n - 1] \]
Linear Recursion as C++ function

// return sum of first n values of array A
int LinearSum(int A[], int n) {
    if (n == 1) {
        return A[0]; // the first value
    } else {
        return LinearSum(A, n-1) + A[n-1];
    }
}
Reversing an Array

**Algorithm** ReverseArray(A, i, j):

*Input:* An array $A$ and nonnegative integer indices $i$ and $j$

*Output:* The reversal of the elements in $A$ starting at index $i$ and ending at $j$

if $i < j$ then

- Swap $A[i]$ and $A[j]$
- ReverseArray($A$, $i + 1$, $j - 1$)

return
Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional parameters that are passed to the method.
- For example, we defined the array reversal method as `ReverseArray(A, i, j)`, not `ReverseArray(A)`. 
Computing Powers

- The power function, \( p(x,n) = x^n \), can be defined recursively:

\[
p(x,n) = \begin{cases} 
1 & \text{if } n = 0 \\
x \cdot p(x, n - 1) & \text{else}
\end{cases}
\]

- This leads to an power function that runs in \( O(n) \) time (for we make \( n \) recursive calls).

- We can do better than this, however.
Recursion vs. iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*

- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions

- Recursion can simplify the solution of a problem, often resulting in *shorter*, more easily understood source code
Problem: Search

- We are given a list of records.
- Each record has an associated key.
- Give efficient algorithm for searching for a record containing a particular key.
- Efficiency is quantified in terms of average time analysis (number of comparisons) to retrieve an item.
Serial Search

● Step through array of records, one at a time.
● Look for record with matching key.
● Search stops when
  ▪ record with matching key is found
  ▪ or when search has examined all records without success.
Pseudocode for Serial Search

// Search for a desired item in the n array elements
// starting at a[0].
// Returns pointer to desired record if found.
// Otherwise, return NULL

... for(i = 0; i < n; i++)
    if(a[i] is desired item)
        return &a[i];

// if we drop through loop, then desired item was not found
return NULL;
Serial Search Analysis

- What are the worst and average case running times for serial search?
- We must determine the O-notation for the number of operations required in search.
- Number of operations depends on \( n \), the number of entries in the list.
Worst Case Time for Serial Search

- For an array of \( n \) elements, the worst case time for serial search requires \( n \) array accesses: \( O(n) \).
- Consider cases where we must loop over all \( n \) records:
  - desired record appears in the last position of the array
  - desired record does not appear in the array at all
Average Case for Serial Search

Assumptions:

1. All keys are equally likely in a search
2. We always search for a key that is in the array

Example:

- We have an array of 10 records.
- If search for the first record, then it requires 1 array access; if the second, then 2 array accesses. etc.

The average of all these searches is:

\[
(1+2+3+4+5+6+7+8+9+10)/10 = 5.5
\]
Average Case Time for Serial Search

Generalize for array size $n$.

Expression for average-case running time:

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

Therefore, average case time complexity for serial search is $O(n)$. 
Binary Search

- Perhaps we can do better than \( O(n) \) in the average case?
- Assume that we are given an array of records that is sorted. For instance:
  - an array of records with integer keys sorted from smallest to largest (e.g., ID numbers), or
  - an array of records with string keys sorted in alphabetical order (e.g., names).
... 
if(size == 0) 
    found = false;
else {
    middle = index of approximate midpoint of array segment;
    if(target == a[middle])
        target has been found!
    else if(target < a[middle])
        search for target in area before midpoint;
    else
        search for target in area after midpoint;
}
Binary Search

Example: sorted array of integer keys. Target=7.

```
[ 0 ] [ 1 ] [ 2 ] [ 3 ] [ 4 ] [ 5 ] [ 6 ]
  3   6   7  11  32  33  53
```
Example: sorted array of integer keys. Target=7.

Find approximate midpoint
Example: sorted array of integer keys. Target=7.

Is 7 = midpoint key? NO.
Example: sorted array of integer keys. Target=7.

Is 7 < midpoint key? YES.
Binary Search

Example: sorted array of integer keys. Target=7.

[0] [1] [2] [3] [4] [5] [6]

3  6  7  11  32  33  53

Search for the target in the area before midpoint.
Example: sorted array of integer keys. Target=7.

Find approximate midpoint
Example: sorted array of integer keys. Target=7.

```
[ 0 ] [ 1 ] [ 2 ] [ 3 ] [ 4 ] [ 5 ] [ 6 ]
  3   6   7   11  32  33  53
```

Target = key of midpoint? NO.
Example: sorted array of integer keys. Target=7.

Target < key of midpoint? NO.
Binary Search

Example: sorted array of integer keys. Target=7.

Target > key of midpoint? YES.
Example: sorted array of integer keys. Target=7.

Search for the target in the area after midpoint.
Example: sorted array of integer keys. Target=7.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td>32</td>
<td>33</td>
<td>53</td>
</tr>
</tbody>
</table>

Find approximate midpoint.
Is target = midpoint key? YES.
/*
* searches for a value in sorted array
* arr is an array to search in
* left is an index of left boundary
* right is an index of right boundary
* value is searched value
* returns position of searched value, if it presents in the array
* or -1, if it is absent
*/
int binarySearch(int arr[], int left, int right, int value) {
    while (left <= right) {
        int middle = (left + right) / 2;
        if (arr[middle] == value)
            return middle;
        else if (arr[middle] > value)
            right = middle - 1;
        else
            left = middle + 1;
    }
    return -1;
}
# Binary Search Implementation

```cpp
#include <iostream>

using namespace std;

int binarySearch(int arr[], int left, int right, int value) {
    while (left <= right) {
        int middle = (left + right) / 2;
        if (arr[middle] == value)
            return middle;
        else if (arr[middle] > value)
            right = middle - 1;
        else
            left = middle + 1;
    }
    return -1;
}

int main(void) {
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n-1, x);
    (result == -1) ? cout << "Element is not present in array" << endl
                    : cout << "Element is present at index " << result << endl;
    return 0;
}
```
```cpp
// binary_search example

#include <iostream>    // std::cout
#include <algorithm>    // std::binary_search, std::sort
#include <vector>       // std::vector

bool myfunction (int i, int j) { return (i<j); }

int main () {
    int myints[] = {1,2,3,4,5,4,3,2,1};
    std::vector<int> v(myints, myints+9);      // 1 2 3 4 5 4 3 2 1

    // using default comparison:
    std::sort (v.begin(), v.end());

    std::cout << "looking for a 3... ";
    if (std::binary_search (v.begin(), v.end(), 3))
        std::cout << "found!\n"; else std::cout << "not found.\n";

    // using myfunction as comp:
    std::sort (v.begin(), v.end(), myfunction);

    std::cout << "looking for a 6... ";
    if (std::binary_search (v.begin(), v.end(), 6, myfunction))
        std::cout << "found!\n"; else std::cout << "not found.\n";

    return 0;
}
```
```cpp
#include <iostream>
#include <algorithm>

using namespace std;

void show(int a[], int arraysize)
{
    for(int i = 0; i < arraysize; ++i)
    
    cout << 't' << a[i];
}

int main()
{
    int a[] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
    int asize = sizeof(a) / sizeof(a[0]);
    cout << "\n The array is : ";
    show(a, asize);

    cout << "\n\nLet's say we want to search for 2 in the array";
    cout << "\n So, we first sort the array";
    sort(a, a + 10);
    cout << "\n\n The array after sorting is : ";
    show(a, asize);
    cout << "\n\nNow, we do the binary search";
    if (binary_search(a, a + 10, 2))
        cout << "\nElement found in the array";
    else
        cout << "\nElement not found in the array";

    cout << "\n\nNow, say we want to search for 10";
    if (binary_search(a, a + 10, 10))
        cout << "\nElement found in the array";
    else
        cout << "\nElement not found in the array";

    return 0;
}```
Binary Search: Analysis

- Worst case complexity?
- What is the maximum depth of recursive calls in binary search as function of $n$?
- Each level in the recursion, we split the array in half (divide by two).
- Therefore maximum recursion depth is floor($\log_2 n$) and worst case = $O(\log_2 n)$.
- Average case is also = $O(\log_2 n)$.
Can we do better than $O(\log_2 n)$?

- Average and worst case of serial search = $O(n)$
- Average and worst case of binary search = $O(\log_2 n)$

Can we do better than this?

YES. Use a hash table!