Analysis of Some Elementry Algorithms

Selection Sort

- Take the first element and compare it with the other members of the array
- Then if the element chosen is bigger than the one found, then it's swapped
- If not, then the contents of the array after that scan stays the same
    - The beginning gets sorted first

When starting a new scan, always assume the first element is from left to right

5 3 2 4 7 8 5 3	min = 5	5 > 3, 3 > 2	swap 5 and 2
2 3 5 4 7 8 5 3	min = 3		
2 3 5 4 7 8 5 3	min = 5	5 > 4, 4 > 3	swap 5 and 3
2 3 3 4 7 8 5 5	min = 4		
2 3 3 4 7 8 5 5	min = 7	7 > 5	swap 7 and 5
2 3 3 4 5 8 7 5	min = 8	8 > 7, 7 > 5	swap 7 and 5
2 3 3 4 5 5 7 8	min = 7		

BIG O ANALYSIS:

WORST CASE SCENARIO Big O – is n^2 (Quadratic time)

$$\begin{array}{ll} & Sn = 1 + 2 + 3 + 4 + \ldots\ldots + n \\ + & Sn = n + (n-1) + (n-2) + (n-3)\ldots\ldots\ldots+1 \\ \hline \end{array}$$

$2Sn = (n+1) + (n+1) \ldots\ldots = n(n+1)$

$Sn = (n(n+1) / 2) - 1$ – remove all constants — n^2

```
Void Selection Sort (int arr[], int n){
        Int i, j, min;
        For (i = 0; i < n - 1; i++){
                if(arr[j] < arr[min]){
                        min = j;
                }
        }
        If (min != i){
                swap(arr[min], arr[i])
        }
}
```

Bubble Sort

- Always start with the first index position and compare it with the next
- If the one chosen is greater than the one found, then swap
- Once swapped, compare the one chosen with the one next
- Once the next found is greater than the one chosen, scan is finished
    - The end gets sorted first

When starting scan, always start from left to right, (like a bubble)

5 3 2 4 7 8 5 3 takes 5, 5 > 3, swap, 5 > 2, 5 > 4, swap, 5 < 7, 7 < 8, 8 > 5, swap, 8 > 3, swap

3 2 4 5 7 5 3 8	takes 3, 3 > 2, swap, 3 < 4, 4 < 5, 5 < 7, 7 > 5, swap, 7 > 3, swap
2 3 4 5 5 3 7 8	takes 2,......5 > 3, swap
2 3 4 5 3 5 7 8	takes 2,......5 > 3, swap
2 3 4 3 5 5 7 8	takes 2,......4 > 3, swap
2 3 3 4 5 5 7 8	takes 2

BIG O ANALYSIS

- Do while it will make n passes → nested loop is also dependent of n
- The first scan you analyize n times, then next n-1

```
void bubble_sort(int arr[], int n){
   int j,k, pos;

   for(k=0; k<n-1;k++){
      pos = 0;
      for (j = 1; j < n-k +1; ++j){
         if(list[j]<list[pos]){
            pos = j;
         }
         swap = list[n -k+1];
         list[n-k+1]=list[pos];
         list[pos]=swap;
      }
      cout<<endl;
      for (int i = 0; i<n; i++){
         cout<<list[i];
      }
   }
}
```

Practice Problems (on attached document)

Asymptotic Upper Bound

ex) number of digits of 2n

- First consider it the power of 10 (ex – 10000, divide by 10, so forth – 5 digits)
- Log base n – +1 will give me the number of digits
- log base 10

ex) the number of times that n can be divided by 10 before dropping below 1.0

Worst case - a n-1, the worst the algorithm can do

Insertion Sort, Linear Search, Binary Search, AND Template Functions

Insertion Sort

- Takes first position and keeps it as i (dependent on the size on n)
- Compare to the next , j , if the next is less than i, then switch
- Keep that i until it is greater than the next number – then switch

```
26 | 18 29 27 30 18 15    26 > 18, swap
18 26 | 29 27 30 18 15
18 26 29 | 27 30 18 15    29 > 27, swap
18 26 27 29 | 30 18 15
18 26 27 29 30 | 18 15   30 > 18, swap, 29 > 18, swap, 27 > 18, swap, 26 > 18, swap
18 18 26 27 29 30 | 15   30 > 15, swap, 29 > 15, swap, 27 > 15, swap, 26 > 15, swap, 18's > 15
15 18 18 26 27 29 30 |
```

```
For (int i =1; size < -1; i++){
        Next = data[i];
        J = i;
        While (j > 0 && next > data[j]){
                Data [j-1] = data[j];
                J–;
        }
        Data[j] = next;
}
```

Worst-case scenario = n^2

- If you insert the key the compare the keys ahead of it

Big O(n)

- If it is already sorted

Linear Search

- Analyze the code – finds the first occurrence of an entry in a list
- Index at which that – that first occurrence is the search key
- Linear search (data, 18)

26 18 29 27 30 18 15

```
For (in i = 0; i < n-1; i++){
        If (key == data[i]){
                Return i;
        }
        Return - 1;
}
```

Best Big O (1)
Worst Big O (n)

- linear function all n entries

<u>Binary Search</u>

- Binary breaks the list in half and analyzes whether the search key is present
- If the data set is sorted – picks last data set as high, first as low, and middle the half of data set.
    - If search key == mid; //then return mid
- If the search key < high, the set between high and mid is cut off -> high = mid-1;
- If search key is > low, then the set between low and mid is cut off low = mid+1
-

```
Low = 0;
High = n -1;
Mid = (low+ high)/2;

While (low < high){
        Mid = (low + high) / 2;
        If (key == data[mid]){
                Return mid;
        }if (key < data[mid]){
                high= mid-1;
        }else
                Low = mid+1;
}
Return -1-low;
```

Data set – 2 3 5 5 7 11 12 17 17 19

KEY = 7

Low	high	low<=high	mid	key==data[mid]	key< data[mid]
0	9	true	4	true	

TERMINATE – FOUND

KEY = 13

Low	high	low<=high	mid	key==data[mid]	key<data [mid]
0	9	true	4	false	false

<u>Key is to the right of mid</u>

Low	high	low<=high	mid	key==data[mid]	key<data [mid]
5	9	true	7	false	true

<u>Key is to the left of mid</u>

Low	high	low<=high	mid	key==data[mid]	key<data [mid]
5	6	true	5	false	true

Key is right of mid

Low	high	low<=high	mid	key==data[mid]	key<data [mid]
6	6	true	6	true	

TERMINATE – FOUND

Key = 9

Low	high	low<=high	mid	key==data[mid]	key<data [mid]
0	9	true	4	false	false

Key is to the right of mid

Low	high	low<=high	mid	key==data[mid]	key<data [mid]
5	9	true	7	false	true

Key is to the left of mid

Low	high	low<=high	mid	key==data[mid]	key<data [mid]
5	6	true	5	false	true

Key is to the left of mid

Low	high	low<=high	mid	key==data[mid]	key<data [mid]
5	4	false			

TERMINATE – CANNOT FIND

Binary search is logarithmic time
BEST CASE – FINDING MIDDLE (BIG O of 1)
BIG O (Log(n)) WORST CASE –
NOT IN LIST, HALFING LIST UNTIL ONE ENTRY, and it's not that number
2^x = n;   log(2^x) = log n;   x = log n;
Template Functions

Write a function that swaps two values

```
void swap (int &x, int &y){
        int temp = x;
        x = y;
        y = temp;
}
u = 5;
v = 7;
Swap (x,y);  //u and v has it's own memory, x is reference to u and v reference to y
```

x overrides temp and gives 5
y overrides the x and gives 7
temp overrides y and gives 5

Parametric polymorphism – a data type and a representation of that data type (ADT)
And make that data type the parameter
template<typename T> //convention is uppercase letter

- Represents any data type, no matter the situation or data type

- In the first call, int is the concrete type for T, String is the second concrete
- Substitution – specialized to handle it any data type

In case of Swap —> void swap (T &x, T&y)

```
                                        T temp = x;
                                        X = y; y = temp;
```

In the case of linear search

```
template <typename T>
Int linearsearch (T data[], T key, int size){
```

What is Recursion (vs. Iteration)
Box Trace Dynamically, Advantages of Pitfall, tail vs non-tail recursion

<u>In addition to using a loop, recursion is another way to solve the problem</u>

- *A recursive solution* – a solution that is defined in terms of the problem itself (smaller version – recursive formulation of the solution)
- Since recursion and iteration solve repetition
    - RecursionADVANTAGE: recursive is very compact (in a more elegant way)
    - RecursionDISADVANTAGE: gloss over the inefficiency/ the formula is in a recursive manner and MORE MEMORY/Stackframe

Recursion - calling itself using the return Class name until the argument is satisfied

- Allows repetition to occur – safer way instead of for loop

<u>Factorial</u>

- N (factorial) is the product of all the integers from n to 1
- In terms of arrangement – a b c (3x2x1 = 6 times)
- 7! Is the same as 7 * 6! (or n (n-1)!)
- If the number is 0, factorial is 0 → otherwise, n * the function-1

In iteration code

```
ans = 1;
cin>>n;
For (for i = 1, i < n; i++){
        ans = ans*n;
}
```

In recursive code

```
Int fact(int n){
        If (n ==0 ){
                Return 1;
        }
        Return n*fact(n-1);
}
```

<u>Stackframe diagram</u>

- Every function calls a stackframe diagram is like a memory

StackTrace Diagram

- To trace out the recursion – two linear, one nonlinear
- Also called the call tree or callstack/boxtrace
- A lot of memory was used – it gaves us an exception

Ex 1) Draw stackframe of fact(5) – given the code (All the lines under fact 5 are in boxes)

Fact (5)

Returns 5*fact(4)

Returns 4*fact(3)

Returns 3*fact(2)

Returns 2*fact(1)

Returns 1*fact(0)

Returns 1;

1 goes back up and returns the return above (back recursion)
1 * 1 = 1 1*2 = 2 2*3 = 6 6*4 = 24 24*5 = … and so forth

ON TEST QUESTION
PART 1 — Gives recursive, draw the stack trace diagram
PART 2 — How many calls were made to the function

- the same as the number of boxes provided by the first call in the box

PART 3 — list the calls in the way they were made

- Fact 5, fact 4, fact 3, fact 2, fact 1

Fibbonacci Sequence - the next number is the sum of the preceding (1 1 2 3 5 8 13 21)

In Iteration Code

```
Ans = 0;
If (n == 1 || n == 2){
        Ans = 1;
}else
        Current = 1;
        Previous = 1;
        For (i = 3; i < n; i++){
                Current = current+previous;
                Previous = current - previous;
                Ans = ans + current;
        }
```

Ex) find 7th value of fibonacci using iteration

Prev	current
1	1
1	2
2	3
3	5
5	8
8	13

In Recursive code

```
Int fib(){
        If (n == 1 || n==2){
                Return 1;
        }else
                Return fib(n-1) + fib(n-2);
}
```

In Recursive

- If the index is 1 or 2, then the return is 1;
- To find any other fibonacci number, add the two preceding numbers

f(5)

Return f(4) + f(3) //compiler does left-right

Return f(3) + f(2) **return f(2) + f(1)**

\ **return 1;**//3 +1 = 4 goes to right **return 1**; //4+1=5

Return (f2) + f(1) **return 1**; //1+2 = 3; so it goes back up to the right side

Return 1; **return 1**; //1+1 = 2; so it goes back up

How many Calls – the amount of times boxed (the bolded part)

List in order

f(5) ,f(4), f(3), f(2), f(1), f(2), f(3), f(2), f(1);

List distinct calls in multiplicity

f(5) – called once

f(4) – called once

f(3) – twice

f(2) – thee times

f(1) – twice

<u>Power Function</u>

x^n → if n = 1, then x^n = x; If n >1, then x = x*x^n-1

Stackframe of 2^6

```
          Pow (2,6) //64
          Return 2*pow(2,5) //32
               return 2*pow(2,4) //16
                    Return 2*pow(2,3) //8
                         return 2*pow(2,2) //4
                              Return 2*pow(2,1) //2
                                    Return 2; //then backtrack
```

Any multiplicities, 6 calls, but listing them is only the second and the second to last

Non-tail vs TAIL Recursive FIND ACTUAL DEFINITION

- TAIL Call to itself as the last statement only once
- TAIL The call to itself any shouldn't have operation outside

Non-tail recursive example

```
fact(int n){
     If (n==0 || n==1){
          Return 1;
     }
}    return n*fact(n-1); //this function is recursive, that calls itself
```

Tail recursive example

```
Int fact(int n, int ans){
     If ( n == 0 || n==1){
          Returns ans;
     }return fact(n-1,n*ans)
} //calls itself once, and the call is the very last statement
```

Purpose

- If you have tail recursive – there are compilers (optimizing)
    - Rewrite the code as an ordinary loop - only uses one stackframe
    - That reason for recruiting or nontail or tail, we are better off rewriting to use less memory

Ackerman function (FIND YT)

```
Int ack(int m, int n){                    //in recursive code
     If ( m == 0){
```

```
                Return n+1;
        }if (n==0){
                Return ack(m-1,1);
        }
        Return ack(m-1,ack(m+1,n))
}
```

Mutual recursion: Define the function of another – it defines what side it’s on

```
static void merge(T data[], T first[], int sizeFirst, T second[], int sizeSecond){
  //Implement this function
        int iFirst= 0, iSecond = 0, j = 0;
        while (iFirst < sizeFirst && iSecond < sizeSecond){
        if (first[iFirst] < second[iSecond]){
                data[j] = first[iFirst];
                iFirst++;
        }else{
                data[j] = second[iSecond];
                iSecond++;
        }
                j++;
        }
                ArrayUtil::arrayCopy(first,iFirst,data,j,sizeFirst-iFirst);
                ArrayUtil::arrayCopy(second,iSecond,data,j,sizeSecond-iSecond);

}
‘
```

```
template <typename T>
static void merge(T data[], T first[], int sizeFirst, T second[], int sizeSecond){
  //Implement this function
        int iFirst= 0, iSecond = 0, j = 0;
        while (iFirst < sizeFirst && iSecond < sizeSecond){
                if (first[iFirst] < second[iSecond]){
                        data[j] = first[iFirst];
                        iFirst++;
                }else{
                        data[j] = second[iSecond];
                        iSecond++;
        }
        j++;
    }
    ArrayUtil::arrayCopy(first,iFirst,data,j,sizeFirst-iFirst);
    ArrayUtil::arrayCopy(second,iSecond,data,j,sizeSecond-iSecond);
}
template <typename T>
inline void SortUtil::bubbleSort(T data[], int size){
     int j,k, pos;

     for(k=0; k<size-1;k++){
     pos = 0;
       for (j = 1; j < size-k +1; ++j){
        if(data[j]<data[pos]){
         pos = j;
        }
        swap = data[size-k+1];
        data[size-k+1]=data[pos];
        data[pos]=swap;
       }
         }
}
template <typename T>
inline void SortUtil::selectionSort(T data[], int size){
    int i, j, min;
    for (i = 0; i < size - 1; i++){
    if(data[j] < data[min]){
        min = j;
    }
    }
    if (min != i){
    swap(data[min], data[i]);
    }
}

template <typename T>
inline void SortUtil::insertionSort(T data[], int size){
    //implement this function
    int next = 0, j = 0,i;
    for (i = 1; i < size; i++){
    next = data[i];
    j = i-1;
    while (j >= 0 && next < data[j] ){
        data[j+1] = data[j];
```

```
                j--;
        }
        data[j+1] = next;
        }
}

template <typename T>
inline void SortUtil::quickSort(T data[], int start, int end){
        if (start >= end){
        return;
        }
        int p = partition(data,start,end);
        quicksort(data,start,p-1);
        quicksort(data,p+1,end);
}

template <typename T>
inline void SortUtil::mergeSort(T data[], int size){
        //implement this function
        T *first, *second;
        if (size > 1){
        first = new T[size/2];
        second = new T[size-(size/2)];
        ArrayUtil::arrayCopy(data,0,first,0,size/2);
        ArrayUtil::arrayCopy(data,size/2,second,0,size-(size/2));
        mergeSort(first,size/2);
        mergeSort(second,size-(size/2));
        merge(data,first,size/2,second,size-(size/2));
        }

}
```

Recursive Sorting - Algorithm Merge Sort

<u>Sorting Problem</u>

- Arranging in some order given a certain permutation
- We can arrange data in whatever order
    - Given a list – numerically from 0 to n
- Want to find a way where d1 >= d2 >= d3 >= d4…

<u>Characteristics to sort</u>

- Stable: whenever I can pair two keys out of the data items to arrange
    - If Two keys are equal, there is no point should they change their relative positions
    - If there are two fours, they shouldn’t swap
- InPlace: the container is in the same place; there shouldn’t be a secondary storage

    - If you sort data within the same container
- Order Optimality: if that class of problem, it’s the best I could do
    - If there was a comparative – if that is the best, it’s order optimal
    - Selection, bubble, and insertion are not order optional,but they are inplace and stable

Merge Sort

- O(nlogn) - asymptotic linear (growth rate is lower)
- Partition in halves until there are groups of 2.
- Each group will order itself then rejoin with previous half and sort until last sort

```
[5,7,12,13]    [4,6,8,12]     //takes these two halves to one until they are sorted
 i               j
[4      ] //move j one
—-------------------------------------------
[5,7,12,13]    [4,6,8,12]
 i                 j
[4, 5 ] //move i one
—-------------------------------------------
[5,7,12,13]    [4,6,8,12]
   i               j
[4, 5,6 ] //move j one
—-------------------------------------------
[5,7,12,13]    [4,6,8,12]
   i                 j
[4, 5,6,7 ] //move i one
—-------------------------------------------
[5,7,12,13]    [4,6,8,12]
     i               j
[4, 5,6,7,8 ] //move j one
—-------------------------------------------
[5,7,12,13]    [4,6,8,12]
     i                 j
[4, 5,6,7,8,12, ] //move i one
—-------------------------------------------
[5,7,12,13]    [4,6,8,12]
        i              j
[4, 5,6,7,8,12,12 ] //move j one
—-------------------------------------------
[5,7,12,13]    [4,6,8,12]
         i             j
[4, 5,6,7,8,12,12,13 ] //move j one
```

Merge Sort cont.

```
mergesort(data[0…n-1]){
        If (n > 1){
                first = data[0,(n-1)/2] //partition into two
                second = data [ (n-1)/2, n-1];
                mergeSort(first);
                mergeSort(second);
        }
        merge(first,second,data); //from left to right and then they override
        //auxiliary function two elements if one is smaller, it advances and moves it's index
}
template <typename T>
static T partition(T data[], T start, T end){
        T pivot, i,j;
        pivot = data[start];
        i = start - 1;
        j = end +1;
        do {
        i = i+1;
        do{
        i = i+1;
        }while(data[i] < pivot);
        j = j-1;
        do{
        j = j-1;
        }while(data[j] > pivot);
        if (i < j){
        swap(data[i],data[j]);
        }
        }while (i < j);
        return j;
}
```

Example of Merge Sort

38 16 27 39 12 27 //original array - checks size of array

//0+5/2 = 2 → *the left entry will always have one more than the right*

First Call → mergeSort[(38,16,27,39,12,27])

Left	**Right**
38 16 27	**39 12 27**

Second Call –> mergeSort([38,16,27]) //the first half is called and merge sort isn't in place

Left1	**Right1**
38 16	**27**

Third Call –> mergeSort([38,16])

Left2	**Right2**
38	**16**

Fourth Call → mergeSort([38]) //the size isn't greater than one, cannot go left go the second call
Fifth Call → mergeSort([16]) //both terminate then it calls to merge the two sorted halves
Sixth Call → merge([38],[16],[38,16])
//when I call, it takes those two sorted and overrides Left1
38 is j and 16 is i //dotted line to indicate
Left1 //overrided as 16 and 31
Seventh Call → mergeSort([27]) //it knows it's sorted, so it falls out
//merge them together – red is the Left
Eighth Call → merge([16,38],[27],[38,16,27])
//what would be the contents of data array after first call to merge?
//this is the contents of the first after first call is arranged, when you make the call, after calling merge, it doesn't alter first and second, the partitions AFTER Left would be same before and after
Ninth Call → mergeSort([39,12,27])

Left3	**Right3**
39 12	**27**

Tenth Call → mergeSort([39,12])

Left4	**Right4**
39	**12**

Eleventh Call → mergeSort([39])
Twelfth Call → mergeSort([12])
Thirteenth Call → merge([39],[12],[39,12]) //after the call it would be 12, 39
//now sorted left half, call the right half of branch (Right 3)
Fourteenth Call → mergeSort([27])
Fifteenth Call → merge([12,39],[27],[39,12,27]) //data represents the right
//then it sorts in red the first right
Sixteenth Call → merge([16,27,38],[12,27,39],[38,16,27,39,12,27])
//Left always goes first → MERGE SORT IS STABLE – does the sort above this
How many calls to merge → 5 calls
//merge has two have two partitions to combine OR n-1 partitions

How many calls to mergeSort → 11 calls
//for each to call to merge, you must have called merge twice. The number calls to merge +1 is sort

The fourth call to merge – is the fourth time it was called, immediately called what the entry is
First entry of second – 27, last entry of second 27

What's the first entry of data immediately called → 12
After the call, the data is already sorted –the contents of that array are already sorted

What would be the first entry of first

What would be last entry of data after the fourth call to emerge

- Identify the fourth call to merge, immediately and override the original, after that call, it only modifies the third array or data – so the first entry is 12
- If it was when it was called, just read what it is, if after → you write the data is sorted (not necessarily sorted, but you answer it like it)

Solving Recurrence Equation

- In dividing, the number of times is log(n) // or log base 2 of n
- Each partition, you combine n elements, of the depths of this tree *n
- Nlogn – thus it is order optional

Template Functions

- Implement merge sort and insertion sort
- Write algorithms to sort integers but now with a type parameter with different data types

ex) linear search

```
Int linsearch(int data[],int size,int item){
        //finds item
        //limited only find an array in integers
        //Principle of least commitment in Software Engineering → instead of committing
        only to working with integers, it designs with any data type
        //this function is limited of type int  – but with any data type to make it templatized
}
```

To templitize

```
template<typename E, typename Q> //two typename parameters

Int linsearch(E data[],int size, E item){ //you can still return an int
Void printPair(E x, Q y); //work to print any data type
```

Quick Sort

A data set→ if you partition it, the first half is <= higher half

- Continue to partition until you have size one
- Create a pivot for each partition call and compare indices i (first) and j (last)
- If pivot > i, then move i → if pivot > j, then swap i and j; if pivot < j, then move j;

```
Quicksort(data[0,.....n-1], i, j){
    //i = 0, j = n-1;
    If i >= j
        return;
    p = partition(data,i,j); //gives me the index such that all entries from i to p <= p to jqu
    quicksort(data,i,p);
    quicksort(data,p+1,j);
}
```

38 16 27 39 12 27

Call 1: qs([38,16,27,39,12,27],0,5) //the array, the first position and the last

Call 2: partition([38,16,27,39,12,27],

//the oracle takes the first index in subarray (pivot) pivot = 38

//all the elements will be <= 38, and right >=

38 16 27 39 12 27

i j pivot = 38 //if the j < then it then swap

27 16 27 39 12 38 move the i then compare with pivot (do the rest)

27 16 27 12 | 39 38 then you have two partitions

Call 3: qs([27,16,27,12,39,38], 0, 3)

//you only wnat to sort the sub array or first parition that is currently there

Call 4: partition([27,17,27,39,38],0,3) //now generating another partition (need 5 in total)

27 16 27 12

i j pivot = 27 //since i isn't less, move j

27 16 27 12 //since 12 is less than 27, swap and move i

i j

12 16 27 27 //considering that a movement there is aleft and a right movement

i j //since they are stuck in the same position, then they don't move

12 16 27 | 27 //paritioned

Call 5: qs([12,16,27,27,39,38],0,2); //write the whole array but only the indices where we sorted

Compares 0 and 2, so that there are two entries so call partition on original data set

Call 6: part([12,16,27,27,39,38],0,2);

12 16 27

i j pivot = 12; //compare if i or j is greater than pivot

12 | 16 27 //such that the pivot is less than the partition above

ij

Call 7: qs([12,16,27,27,39,38],0,0); //checks 0 and 0 but there is only entry

Call 8: qs([12,16,27,27,39,38],1,2) //1 1!> 2 so it calls

Call 9: part([12,16,27,27,39,38],1,2);

16 27

i j pivot = 16; //compare i and j to pivot

16 | 27

Call 10: qs([12,16,27,27,39,38],1,1) //knows that the sub array contains one entry and returns

Call 11: qs([12,16,27,27,39,38],2,2) //it returns again

Call 12: qs([12,16,27,27,39,38],3,3) //returns

Call 13: qs([12,16,27,27,39,38],4,5) //considering that 4 !> 5, then partition

Call 14 part([12,16,27,27,39,38],4,5)

… 39 38 …

i j pivot = 39; compare and move

39 38

i j

38 | 39

//since it isn't left than pivot, move j and j is on 38 and compare to 39, then we are stuck but we dont' stop because the indices (j and i) have overlapped

Call 15: qs([12,16,27,27,38,39],4,4) //returns
Call 16: qs([12,16,27,27,38,39],5,5) //returns
5 Calls to partition, and 11 to Qs (5*2 +1)

Question:
What is the first enty of data array during the fourth call when the fourth call to qs was made → 12
Last entry is → 38

- **qs Contents of data array**
- **partition the contents data array of after**

Doesn't guarantee optimal performance –worst case scenario n^2

If i have template functions in the cpp or h in the main you #include junk.cpp
You still #include in the cpp for the h, but for main you include the implementation
How to measure run time

- Goal of project → you have two functions that sort data insertion and merge
- If i take the same data and put it in two sort (merge sort is shorter than insertion) but insertion is way faster and holds less memory
- Table of both different array sizes,plot a graph → because insertion is quadratic, merge is log

- Empirical algorithmics → analyze algorithms using experimental than mathematical

Measuring runtime using chrono library

- Auto start = high_resuluton_clock::now() //time is measured in clock ticks
- //code
- Auto elapsed //gest during amount of ticks and

To convert it actual time
Long duration = durantion_cast<nano>

Appendix Namespaces (data abstraction/data structures/abstracts Data Types)

<u>NameSpace</u>

- Motivation → if it's a big code with several functions, some of those functions have the same name (obvious conflict)
- It provides a mechanism to group functions a given name
- Definition: Construct to group definitions, functions, etc.
- By default → c++ has a global namespace (using namespace std)
    - All of the basic functions (cout,cin) → don't have to do std::cout
- But if you have three different namespaces with similar features in each, you need to specify which one you are calling

        - NS1::f1;
    - Even if there is a unique function, you still have to call the namespace
    - ONLY if you say using namespace NAME;
        - using namespace ns1;
- Works exactly like an ADT
- You can also use an alias as a namespace
    - namespace foolish = NS1;
    - foolish::f1(); //will print out whatever

<u>To define Namespace</u>

- namespace NSName{

  feature{

  }

  } //cannot have two same features in the same namespace

Data Structure →

- Way of organizing data to make it easily accessible

ADT (Abstract Data Type) →

- Consists of data structure + set of functions to manipulate the collection of data
- Implementation (define the functions), header/interphase/API, client code

a wall the use of the application from implementation → mimic in real world

Friend Classes and Functions, Reference/Constant Reference Parameters, Overloading Prefix / PostFix / overloading bracket and ()

<u>Access specifiers</u>

- Private → accessible within class
- Public → accessible within and outside class
- Classes allow data members private and functions public → encapsulate data

<u>Friend</u>

- Function outside of the class but takes a variable of the object of that class
- Rely on assessor functions or methods that are private
- Nonmember functions that has private member access
    - The prototype has to be within it AND keyword Friend infront of the class
- It avoids more code and it’s way more easier → it’s a combination of nonmember and member functions
- ex) Class point
    - Has private integers coordinates
    - Has public constructors and assessors (Getters and setters)

      <u>SENARIOS (nonmember, member, friend)</u>

- Defining a function midpoint that is non-member
  - Point mid (const Point p1, const Point p2)
    - Return Point((p1.getX()+P2.getX())/2, (P1.getY()+P2.getY())/2);
- If it was a member function → mid(cont Point &p) const; return point
  - Point Point::mid (const &p) const;
    - Return Point ((x+p.x)/2,(y+p.y));
- Friend function → friend Point mid(const Point&p1, const Point&p2);
  - Return Point(p1.x+p2.x/2,p1.y+p2.y/2);
- If it was in main
  - Point P1(1,4) , Point P2(3,2);
  - Non member → Point P3 = mid(p1,p2)
  - Member → P1.mid(P2);

Friend Classes

- ex) Bank Account – has private type of USMoney balance and public getter
  - It can bal.getDollars() because it's outside of the USMoney class, but it has a object type "balance" to get functions
  - To make it a friend → grant friendships from another class
    - Class USMoney
      - friend class BankAccount; //granting to another class than a function
      - //within bankaccount, has anything private
    - In this case you do → return balance.dollars // the money class given permission to use any private things
  - **IF class A grants friendship to class B → class B has access anything that is private to class A**
- ex) Class Traingle to Acess Pointer Clas
  - Private → Point p1,p2,p3; //this is accessible with friend
  - Public → double distance(const Point&p1,const Point&p2); //distance btw 2 point
    - Double Triangle::distance(const Point &p1, const Point&p2);
      - return sqrt(pow(p1.x-p2.x,2)+pow(p1.y-p2.y,2));
    - **If a class A granted friendship to another class B, class B also granted access to all the members within that class A**
    - **Typically if instance variables of class B are objects of class A**
  - In the pointer class
    - Public
      - friend Class Triangle;

Const reference parameters within functions

ex) Point p1 (1,2);
Point p2 (3,5);
Point P3 = mid(p1,p2);
mid (const Point&p1,const Point&p2);

- Significance → since you wnat to change the value temporarily, since they are named the same, in the mid function you grant access to memory value that isn't related

- P1 pointer will take that address as the value of whatever you set it or return in mid function
    - In some cases, you can have copies in memory

Overloading Operators

- x = 5; → cout<<x++<<" "<<++x; // output : 5 6
    - reads cout from left to right;

Prefix

- Overloading the operator ++ for BEFORE so if cout<<++m it changes cents
- USMoney operator++(); // doesn't take parameters, in US Money
- USMoney USMoney::operator++();
    - cents++;
    - dollars += cents / 100;
    - cents = cents % 100;

Postfix

- Overloading the operator ++ for AFTER, cout<<m++
- USMoney operator++(int dummy); //if it has a parameter, it means postfix
- USMoney USMoney::operator++(int dummy);
    - int d = dollars, c = cents; //saving the current value before any increment
    - cents++;
    - dollars += cents / 100;
    - cents = cents % 100;
    - return USMoney(d,c); //returning original, it will use original in expression and then do these updteas → don't need to do any amount, just modify the original

ex) given post fix and prefix functions → m(3,72) in MAIN

- cout<m++; 3.72
- cout<<++m; 3.74
- cout<<m++; 3.74
- cout<<m; 3.75
- cout<++; 3.76

[ ] operator

- Int operator [] (int index);
    - // Allows to access x or y or to modify them;
    - //when you have a point p1[0] → want to get x, p[1] → get y;
    - //throw an exception if values in [ ] is not 0 or 1
- int Point::operator [] (int index)
    - if index < 0 || index > 1 → throw invalid_argument ("out of bounds");
    - if index == 0; → return x; //modifying x the actual point
    - if index == 1; → return y;
- Point p(3,5) → p[0] = 9 → changes y coordinate to 9;

( ) operator

- Same as [ ]
- If it is in money

Project

- arrayUtil.cpp and arrayutil.h are not modified
    - How to use them
    - Array copy →
        - Double* data = new double[10];
        - Double* first = new double[5];
        - Double*second = new double[5];
    - Copies first five into data and the second into data
    - arrayC(src is the big data, starting index for data, first, starting index j in other one,size of first)
        - (data,0,0, first,5)
        - For second = (data,5,0, second,5)
    - GetRandArray →
        - double* dl =new double[100]; //populate array with random doubles
        - getRandArray(d1,100) //fills the array with random numbers
    - Sorted Array →
        - Same thing with getrand
        - Bool ascending true and descending false
- In excel sheet
    - Measure runtime for

Template Classes

Ex) Pairs of anything (bool,int)

```
#ifndef PAIR_H
#define PAIR_H

Using namespace std;
Template <typename T>
class Pair{
        Private:
                T first;
                T second;
        public:
                Pair();
                Pair(T f, T s);
        T getFirst() const;
        T getSecond() const;
        Template <typename U>
        friend void swap(Pair<U>& p);
```

```
    //create a template function of a template class, if it has to have separate template front
    he class itself – if didn’t revolve around any other template parameters, then it can be a
    basic template class
};

#ednif
```

In cpp

```
#include “Pair.h”
Pair<T>::Pair();
Pair<T>:: Pair(T f, T s){
    first = f;
    second = s;
}
Template <typename T>
T Pair<T>::getFirst() const{
    return first;
}
Template <typename T>
 T Pair<T>::getSecond() const{
    return second;
}

Template <typename U>
friend void swap( Pair <U>& p){
    U tmp;
    Tmp = p.second;
    p.second = p.first;
    p.first = tmp;
}
```

Will give us an big error

- If you program normally – you #include .h in cpp and then in main
- If you have templates, you only #inlude cpp in main instead of .h

```
#include iostream
#include “Pair.cpp” //the cpp file will still #include the .h
//cpp file doesnt’ know how to compile those other codes
Using namespace std;
```

Friend Template Classes of template classes

- Point class → points in a triangle (Point v1,point v2, Point v3);
    - In point class grants friendship to the triangle class
    - In triangle, allows class to do v1.getx instead of v1.getx();

```
-   Point{
    Private: //to only grant friendship to another class, it has to be granted in the private
section
            friend class Triangle;
    }
```

If it was granting friendship to a template class

```
Point{
    Private: //given the point class has a different template name → like T
            T first;
            T second;
            Template <typename U>
            Friend class Name;
}
in Main
Int main(){
    Point <int> p1 (3,2);
    Point <int> p2; // it knows that even intialized of type int
}
```

How to templitize more than one in the same object

```
In .h
template<typename S, typename T>
private:
    S first
    T second;
```

```
In .cpp
Pair<S,T>::Pair(); //and so on
Template <typename S, typename T>
T Pair <S,T>:
In Main
```

STACK ADT – Basic Operations and An Extensible Parametric / Stack ADT implement.

<u>Stack Abstract Data Type</u>

- Like a stack of books, the last item that I pushed was the first that I can pop
- LAST IN FIRST OUT or LIFO
  - Linear data structure → provides to create, push, and pop to modify
  - Access functions top, empty, size (doesn't modify stack)
- Destructive function → deallocates memory into the system

    - Write an exception class or customized class

Extensible → elastic, or it doesn't have a fixed size (based on the amount of items)

- Don't waste memory or allocate
- USE POINTERS

Stack Exception

- Anytime needed to report user that something is problematic
- Customized

StackException.h

```
Class StackException{
        private:
                String message; //store reason why problem if occurred
        public:
                StackException(String msg); // default to create exception object
                String what() const;  //call to retrieve the message
}
```

StackException.cpp

```
StackException::StackException(String msg){
        message = msg;
}
String StackException::what() const{
        return msg;
}
```

If it occurs → throw StackException("   "); //

In try catch → you catch (const StackException)

```
        cout<<e.what()<<endl; //then it creates an object of an exception
```

You define the node class within the stack class

Stack<T>

```
        int length; //tells how many items in stack or metadata field
        Head; //node or Node <T> or NULL initially
```

Node<T>

- T data; //store in data field
- Next; //node or Node<T> which points to the

Push and pop

- When you call push, it instantiate or creates an node object

```
        S //when it starts
Length { 0      }
```

Head   { NULL }

If you push

```
length++;
Head and next{  4       }
                    { NULL}
```

Auxiliary functions

```
size()
          Return length;
```

—---

```
top ()
          If (!empty()
          Return head->data;
```

```
template <typename T>
Stack<T>::Stack(){
          length = 0;
          head = nullptr;
}

Template <typename T>
Stack<T>::~Stack(){ //deallocate all the nodes //write the function from scratch
          Node<T>*tmp;
          For int i;
          for (i = 1; i<=length){
                    tmp = head;
                    head = head->next;
                    Delete tmp;
                    //delete all the nodes
                    Head = nullptr;
                    length = 0;
          {
}
Template <typename T>
Void Stack<T>::push(T item){
          Node<T>* m = new Node(nn)
          nn → next = head;
          Head = m;
          length++;
}
Template <typename T>
Void Stack<T>::pop(){
    if (length == 0){
        throw StackException(“POP…..”)
    }

}
```