Part 1:

Search Product

```
public boolean searchForProduct(Product p,Company c){ // he or she can sea
    vector<Branch> branches = c.getBranches();
    for(int branchIndex=0;branchIndex<branches.getUsed();branchIndex++){
        if( branches.at(branchIndex).getProducts().isAvailable(p) != -1){
            return true;
        }
    }
    return false;
}
```

getBranches () ==> Θ (1)

GetUsed() ==> Θ (1)

GetProducts() ==> Θ (1)

IsAvailable(p) :

```
public int isAvailable(E value){
    for(int i=0;i<used;i++){
        if(arr[i].equals(value)){
            return i;
        }
    }
    return -1;
}
```

best case Θ (1)

Worst case Θ (n)

General case O(n)

Equals' asymptotic notation

```java
public boolean equals(Object o) {
    if (this == o)
        return true;
    if (o == null)
        return false;
    if (!(o.getClass().equals(getClass()))){
        return false;
    }
    Product product = (Product) o;
    // field comparison
    return (product.model == model && product.color==color);
}
```

I could not find the Class's equals method, but it works same with == operator. I guess its asymtotic notation is Θ (1).

Equals method's asymptotic notation : Θ (1)

So, SearchForProduct's asymptotic notation is O(n)

Add Product:

```java
public void addProduct(Product p)throws BranchEmployeeDoesNotHaveAuthority{
    if(branch == null){
        throw new BranchEmployeeDoesNotHaveAuthority();
    }
    branch.getProducts().push_back(p);
}
```

getProducts() ==> Θ (1)

push_back ==> Θ (1)

Add Product's asymptotic analysis = Θ (1)

Push_back:

```
public void push_back(E value){
    if(arr.length == used){
        capacity *= 2;
        Object[] temp = (E[])new Object[capacity];
        for(int i=0;i<arr.length;i++){
            temp[i] = arr[i];
        }
        arr = temp;
    }
    arr[used++] = value;

}
```

Best Case Θ (1)

Worst Case Θ (n)

Amortized Running time ==> Θ (1)

So, AddProduct's asymptotic notation is **Θ** (1)

Remove Product:

```
public boolean removeProduct(Product p) throws BranchEmployeeDoesNotHaveAuthority{
    if(branch != null){
        return branch.getProducts().remove(p) != null;
    }
    throw new BranchEmployeeDoesNotHaveAuthority();
}
```

GetProduct() == > Θ (1)

Remove:

```
public E remove(E e){
    for(int i=0;i<used;i++){
        if(arr[i].equals(e)){
            delete(i);
            return e;
        }
    }
    return null;
}
```

Best case Θ (1) + delete(i) ==> O(n)

Worst Case Θ (n)

General Case ==> O(n)

Delete:

```java
public void delete(int index){
    if(index<used){
        for(int i=index;i<arr.length-1;i++){
            arr[i] = arr[i+1];
        }
        pop_back();
    }
    else{
        System.out.println("Invalid index");
    }
}
```

For loop’s best case is Θ (1) , it happens when the last element is deleted

For loop’s worst case is Θ (n) , it happens when the first element is deleted.

Pop_back:

```java
public void pop_back(){
    if(used > 0 ){
        used--;
        if(used <= (capacity/4)){
            capacity /= 2;
            Object[] temp = (E[])new Object[capacity];
            for(int i=0;i<used;i++){
                temp[i] = arr[i];
            }
            arr = temp;
        }
    }
}
```

Best case Θ (1)

Worst case Θ (n)

Amortized running time ==> Θ (1)

So, delete’s asymptotic notation is O(n)

So, removeProduct’s asymptotic notation is O(n)

AskForProductNeed:

```java
public boolean askForProductNeed(Product p,int branchIndex)throws IndexOutOfBoundsException{
    if(branchIndex >= branchesOfCompany.getUsed()){
        throw new IndexOutOfBoundsException();
    }
    if(branchesOfCompany.getUsed()>branchIndex){
        if(branchesOfCompany.at(branchIndex).getProducts().isAvailable(p) == -1){
            addProduct(p, branchIndex);
            return true;
        }
    }
    System.out.println(p+" has not added to "+(branchIndex+1)+". branch because there is already");
    return false;
}
```

IsAvailable() == > O(n)

AddProduct ==> Θ (1) (amortized running time)

So, AskForProductNeed’s asymptotic notation is O(n)

Additionally I have another report ,I want to attach it here too , I could not decide which one is better.

```
public boolean searchForProduct (Product p, Company c){

    Vector< Branch > branches = c.getBranches(); // Θ(1)
                                     └> only returns vector of branches
    for( int branchIndex=0; Θ(1)
         branchIndex < branches.getUsed() ;//Θ(1) -> returns number of branches
         branchIndex++){ //Θ(1)
        if(branches.at(branchIndex). getProducts(). isAvailable(p)!=-1){
                    returns the branch      returns product       } O(n)
                    at the given index      of branch
                    Θ(1)
            return true; //Θ(1)
        }
    }
    return false; //Θ(1)
}
```

Best case Θ(1), Worst Case Θ(n) ⇓ O(n)

```
public int isAvailable (Product p){
    for(int i=0; i< used; i++){ // Θ(1)+Θ(1)+Θ(1)=Θ(1)
        if(arr[i].equals(p)){ //Θ(1)
            return i; // Θ(1)
        }
    }
    return -1; Θ(1)
}
```

Bestcase Θ(1), Worst Case Θ(n)

isAvaible's notation is O(n)

```
public boolean equals (Object o){
    if(this==o)     //Θ(1)
        return true; //Θ(1)
    if(o==null)     //Θ(1)
        return false; //Θ(1)
    if(!(o.getClass() == getClass())){//Θ(1)
        return false; //Θ(1)
    /* I do not know implementation of
    Class class' equals method, I have
    converted it into ==. */
    Product product = (Product) o; //Θ(1)
    return( product.model == model //Θ(1)
            &&
            product.color == color); Θ(1)
```

Θ(1)

```
public void addProduct (Product p) throws BranchEmployeeDoesNotHaveAuthority {
        if( branch == null) {
              throw new BranchEmployeeDoesNotHaveAuthority();
        }
        branch.getProducts().push_back(p);
```

$\Theta(1)$

$\Theta(1)$ ↳ Amortized time complexity $= \Theta(1)$

```
public void push_back (E value)
      if (arr.length == used)
            capacity *= 2
            Object[] temp = (E[]) new Object[capacity];
            for (int i = 0; i < arr.length; i++)
                  temp[i] = arr[i];
            }
            arr[i] = temp[i]
      }
      arr[used++] = value;
}
```

$\Theta(n)$

Best case $\Theta(1)$

Worst Case $\Theta(n)$

Amortized Running time $\Theta(1)$

```
public boolean removeProduct (Product p) throws BranchEmployeeDoesNotHaveAuthority {
      if (branch != null)   // Θ(1) → Θ(1)
            return branch.getProducts(). remove(p) != null;
      throw new BranchEmployeeDoesNotHaveAuthority();
```

$O(n)$

$O(n)$

Best $\Theta(1)$
Worst $\Theta(n)$

```
public void delete(int index) {   → O(n)
      if (index < used) {
            for( int i = index ; i < arr.length-1; i++) {
                  arr[i] = arr[i+1];
            }
            pop_back(); // O(n)
      }
      else
            System.out.println("Invalid index"); // Θ(1)
}
```

best $\Theta(1)$, worst $\Theta(n)$, $O(n)$

```
public E remove (E value) {
      for(int i=0; i < used; i++) {
            if(arr[i].equals(value)) {   Θ(1)
                  delete(i); // O(n)
                  return value; // Θ(1)
            }
      }
      return null; // Θ(1)
}
```

```
public void pop-back(){   -> best case Θ(1)  > O(n)
                             worst case Θ(n)
    if(used>0){ //Θ(1)
        used--; //Θ(1)
        if( used <= (capacity/4)){ //Θ(1)
            capacity /= 2; //Θ(1)
  Θ(1) <- Object []temp = (E[]) new Object[capacity];
            for(int i = 0; i< used; i++){
                temp[i] = arr[i]; //Θ(1)      } Θ(n)
            }
            arr = temp; //Θ(1)
        }                                      Θ(n)
    }
}

public boolean askForProductNeed(Product p, int BranchIndex) => O(n)
                                   throws IndexOutOfBounds{
    if(branchIndex >= branchesOfCompany.getUsed()){
        throw new IndexOutOfBounds();
    }
    if(branchesOfCompany.getUsed() > branchIndex){       -> O(n)
        if(branchesOfCompany.at(branchIndex).getProducts().isAvailable(p)==-1)
            addProduct(p, branchIndex); //amortized Θ(1)
            return true; // Θ(1)
    }
    }
    return false; //Θ(1)
```

Part 2:

A) Explain why it is meaningless to say: “The running time of algorithm A is at least O(n2)”.

Answer:

Big O notation shows the upper bound of the running time of the algorithm. So, the algorithm can not take more time than the function of the Big O notation. When we say that “The runnign time of algorithm A is at least O(n2)” , it seems like the algorithm can take longer time than n^2. But it can not because of the Big O notation.

B) Let f(n) and g(n) be non-decreasing and non-negative functions. Prove or disprove that:

max(f (n), g(n)) = Θ (f(n) + g(n)).

Answer:

Addition of the the functions returns the maximum of them. The left side also returns the max of the functions and Θ notations nailed the running time to within a constant factor above and below. Asymptotic notation do not care about constants and the lower terms, so theta notation returns the max of f(n) and g(n) too. So the equation holds.

C)Are the following true? Prove your answer.

I. 2n+1 = Θ (2n)

Answer:

2^(n+1) is equal to the 2.2^n. Because of that the constants can be ignored, we can say that 2n+1 is equat to the Θ (2n).

II. 2^(2n) = Θ (2^n)

Answer:

2^(2n) = 4^n

So 4^n is greater than 2^n asymptotically, and theta notation gives a tight bound. So the equation does not hold. 2^2n is equal to the Θ(4^n) which is way more greater than Θ(2^n).

III. Let f(n)=O(n2) and g(n)= Θ (n2). Prove or disprove that: f(n) * g(n) = Θ (n4).

Answer:

f(n) can be any functions which is lower than or equal to the k*n^2. Because of that theta notation shows a tight bound around the function, if the f(n) is very small function , the multiplication could be out of the tight bound around n^4. So the equation does not hold.

Part3:

$\log(n) < (\log n)^3$

$\log$ $(\log n)^3$ vs $\sqrt{n}$

$\hookrightarrow \log(\log n)^3$ $\quad \log\sqrt{n}$

$3\log(\log n) < \frac{1}{2}\log n$

$\boxed{(\log n)^3 < \sqrt{n}}$

$\sqrt{n} < n^{1.01}$ $\quad \sqrt{n} < n(\log n)^2$

$n^{1.01}$ vs $n(\log n)^2$

$\lim_{n\to\infty} \frac{n^{1.01}}{n(\log n)^2} = \infty$

$\boxed{n^{1.01} > n(\log n)^2}$

I checked it on internet

$n^{1.01}$ vs $5^{\log_2 n}$

$\log n^{1.01}$ $\quad \log 5^{\log n}$

$1.01 \log n < \log n . \underbrace{\log 5}_{2,...}$

2^n vs $5^{\log n}$

$\hookrightarrow \log 2^n$ $\quad \log n . \log 5$

$n . \cancel{\log 2}$ $\quad \log n . 2,...$

$n > \log n . 2,...$

$2^n > 5^{\log n}$

&&

$n2^n > 2^{n+1} > 2^n$

$n2^n$ vs 3^n

$\log n2^n$ $\quad \log 3^n$

$\log n + \log 2^n$ $\quad n\log 3$

$\log n + n\log 2$ $\quad n\log 3$

$n + \log 2 < n\log 3$

$\boxed{n2^n < 3^n}$

CamScanner ile 1

$\log n < (\log n)^3 < \sqrt{n} < n(\log n)^2 < n^{1.01} < 5^{\log n} < 2^n < 2^{n+1} < n2^n < 3^n$

Part 4:

1)

```
FindMinValue ( A ) {
    min= A.get (0) // Θ(1)                          Θ(1) +Θ(n)+Θ(1)= Θ(n)
    for ( i=1 to n) {
        if ( A.get(i) < min){//Θ(1)
            min =A.get(i)  //Θ(1)                   Θ(n)
        }
    }
    return(min) // Θ(1)
}
```

2)

```
double median(Arr, n){
    bigger = lower = same = 0
    if(n % 2 == 1){
        for(i = 0 to n){
            lower = same = 0
            for(j = 0 to n){
                if(Arr.get(i) > Arr.get(j)){
                    lower++
                }
                else if(Arr.get(i) == Arr.get(j)){
                    same++
                }
            }
            if(lower == n/2){
                return Arr.get(i);
            }
            else if( lower < n/2 &&
                lower+same > n/2){
                return Arr.get(i)
            }
        }
    }
```

$\Theta(1)$

$\Theta(1)$

$\Theta(1)$

Best = $\Theta(1)$

Worst = $\Theta(n^2)$

$\Downarrow$

$O(n^2)$

General case => O(n^2) for arrays has odd number of entries.

$O(n^2)$

Best = $\Theta(1)$
Worst = $\Theta(n^2)$
$\Downarrow$
$O(n^2)$

$\Theta(n)$, $\Theta(1)$, $\Theta(1)$, $\Theta(1)$

```
}
else{
    median1Found = median2Found = false
    index1 = size/2 -1 , index2 = size/2
    for(i=0 to n){
        lower = same = 0
        for(j=0 to n){
            if(Arr.get(i) > Arr.get(j)){
                lower++;
            }else if(Arr.get(i)==Arr.get(j)){
                same++;
            }
        }
        if(median1Found ==false){
            if((lower == index1) ||
               (lower < index1 && lower+same > index1)){
                median1 = Arr.get(i);
                median1Found = true
            }
        }
        if(median2Found == false){
            if((lower == index2) ||
               (lower < index2 && lower + same > index2)){
                median2 = Arr.get(i)
                median2Found = true
            }
        }
        if( median1Found && median2Found){
            return(median1 + median2)/2.0;
        }
    }
}
return -1; // Θ(1)
}
```

Best Case Θ(1)

Worst Case Θ(n^2)

General case ==> O(n^2)

Median algorithm’s notation is O(n^2)

3)

```
int[] twoSum (A, sum){
    for( i=0 to n) {
        for( j=0 to n){
            if(A.get(i)+ A.get(j)== sum) {
                int[] arr = { i, j}
                return arr;
            }
        }
    }
    return (null)
}
```

Best case $\Theta(1)$
Worst case $\Theta(n^2)$
General => $O(n^2)$

4)

```
ArrayList merge (Arr1, Arr2){
    i=j=0
    ArrayList merged;
    for(int k=0; k< 2*n; k++){
        if( j==n ||( i<n && Arr1.get(i)<Arr2.get(j)) { Θ(1)
            merged.add(Arr1.get(i)// Θ(1)
            i++  // Θ(1)
        }
        else{
            merged.add(Arr2.get(j)) // Θ(1)
            j++ // Θ(1)
        }
    }
    return (merged)//Θ(1)
}
```

$\Theta(2n)=\Theta(n)$

Part 5:

Part 5:

	Time Complexity	Space complexity
a)	$\Theta(1)$	$\Theta(1)$
b)	$\Theta(n)$	$\Theta(1)$

c)

$\Theta(\log(i)*n)$ $\Big\{$ `for(int i=0 ;i<n;i++)`

$\Theta(\log(i))$ $\Big\{$ `for(int j=1; j<i ; j*=2)`

`printf("%d", array[i] * array[j])` $\}$ $\Theta(1)$

c)	$\Theta(n\log(n))$	$\Theta(1)$
d)	$\text{if}(\overbrace{p\text{-}2(array,n)}^{\Theta(n)} > 1000)$	$\theta(1)$

$p\text{-}3(array,n)$ $\}$ $\theta(n\log(n))$

else $\text{printf}("\%d", \underbrace{p\text{-}1(array)}_{\Theta(1)} * \underbrace{p\text{-}2(array,n)}_{\Theta(n)})$

Best case

$\Theta(n) + \Theta(n) = \Theta(n)$

Worst case

$\Theta(n) + \Theta(n\log(n))$

$\Longrightarrow O(n\log(n))$