THERE IS A HIERARCHY OF ASYMPTOTIC GROWTH RATES

$\Theta(1)$ {CONSTANT FUNCTIONS}

$\Theta(\lg n)$

$\leftarrow$ $\Theta(\sqrt{n})$

$\Theta(n)$

$\Theta(n \lg n)$

$\leftarrow$ $\Theta(n^{1+\varepsilon})$ $0<\varepsilon<1$

$\Theta(n^2)$

$\Theta(n^2 \lg n)$

$\Theta(n^3)$

$\vdots$

$\Theta(2^n)$

$\leftarrow$ $\Theta(b^n)$ $2<b<3$

$\Theta(3^n)$

$\vdots$

$\Theta(n!)$

$\vdots$

NOTE ALL LOGS ARE THE SAME, ASYMPTOTICALLY SPEAKING SINCE IF $a>1$ AND $b>1$ THEN

$$\log_a(n) = \Theta(\log_b(n))$$

THIS FOLLOWS FROM THE WELL KNOWN CONVERSION FORMULA

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$$

$$\therefore \quad \log_b(n) = \log_b(a) \cdot \log_a(n) = \text{const} \cdot \log_a(n).$$

CONSIDER FOUR ALGORITHMS: A, B, C, D WHOSE RUN TIMES (ANY CASE) ON INPUT OF SIZE n ARE

$$
\begin{aligned}
A &: n^2 \\
B &: 10n^2 \\
C &: 10n^2 + 2n + 100
\end{aligned}
\quad \Bigg\} \ \Theta(n^2)
$$

$$D : 1000n + 10{,}000 \quad \} \ \Theta(n)$$

D IS SUPERIOR FOR LARGE n, AND WORST FOR SMALL n. A, B, AND C ARE CONSIDERED EQUIVALENT. THE LOWER ORDER TERMS IN C ARE NEGLIGIBLE FOR LARGE n, AND A AND B CAN BE EQUALIZED BY RUNNING B ON A MACHINE WHICH IS 10 TIMES FASTER.

THUS ASYMPTOTIC GROWTH RATE OF AN ALGORITHM'S RUN TIME PROVIDES A MACHINE INDEPENDENT MEASURE OF ITS EFFICIENCY.

THIS ALSO EXPLAINS WHY ITS OK TO COUNT ONLY THE BASIC OPERATIONS WHEN ANALYZING AN ALGORITHM. IF WE WERE TO COUNT THE COST OF PERIPHERAL OPERATIONS, THE DIFFERENCE WOULD BE SOME ADDITIONAL LOWER ORDER TERMS, AND POSSIBLY A CHANGE IN THE COEFFICIENT OF THE HIGHEST ORDER TERM

SEQUENTIAL SEARCH WOULD CHANGE FROM n TO $an + b = \Theta(n)$.

BINARY SEARCH WOULD CHANGE FROM $\lfloor \lg n \rfloor + 1$ TO

$$a(\lfloor \lg n \rfloor + 1) + b = \Theta(\lg n)$$

- a: COST OF OPS. INSIDE LOOP.
- $(\lfloor \lg n \rfloor + 1)$: # OF LOOP ITERATIONS
- b: COST OF OPS OUTSIDE LOOP

HOW SHOULD WE CHOOSE WHICH OPERATION TO CONSIDER AS BASIC AND WHICH OPERATIONS TO CONSIDER AS PERIPHERAL?

GENERALLY, WE SHOULD LET THE BASIC OPERATION BE SOME STEP INSIDE THE INNERMOST LOOP FOR ITERATIVE ALGORITHMS

Ex.

```
1   void wasteTime(int n) {
2       int i, j, k
3       waste d units OF TIME
4       for (i=0; i<n; i++) {
5           waste c
6           for (j=0; j<n; j++) {
7               waste b
8               for (k=0; k<n; k++) {
9                   waste a
10              }
11          }
12      }
13  }
```

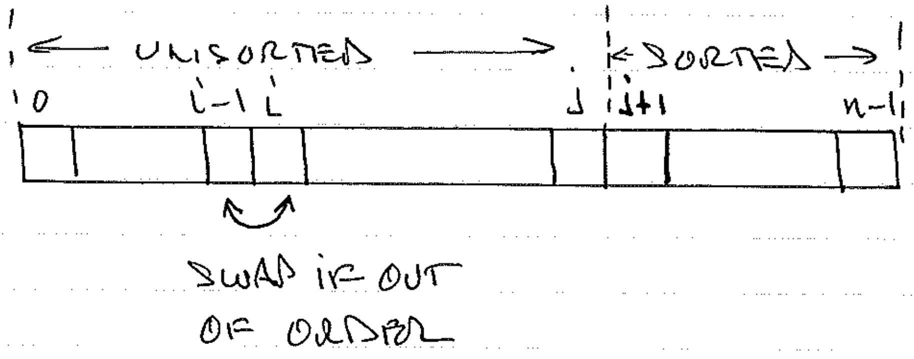
RUNTIME $T(n) = an^3 + bn^2 + cn + d = \Theta(n^3)$. WE MIGHT JUST HAVE WELL CONSIDERED LINE 9 TO BE BAROMETER OPERATION AND COUNTED THAT.

ANALYSIS OF RECURSIVE ALGORITHMS IS DEALT WITH EXTENSIVELY IN CMPS 101 AND CMPS 102.

GENERALLY ONE CHOOSES SOME OPERATION WHICH IS PERFORMED ON EVERY INVOCATION AS BASIC OPERATION. THE RUN TIME THEN ESSENTIALLY THE RECURSION DEPTH.

PROBLEM: SORT AN UNORDERED ARRAY IN INCREASING ORDER.

```
void BubbleSort(int* A, int n) {
    int i, j, temp;
    for (j = n-1; j > 0; j--) {
        for (i = 1; i <= j; i++) {
            if (A[i] < A[i-1]) {
                temp = A[i];
                A[i] = A[i-1];
                A[i-1] = temp;
            }
        }
    }
}
```



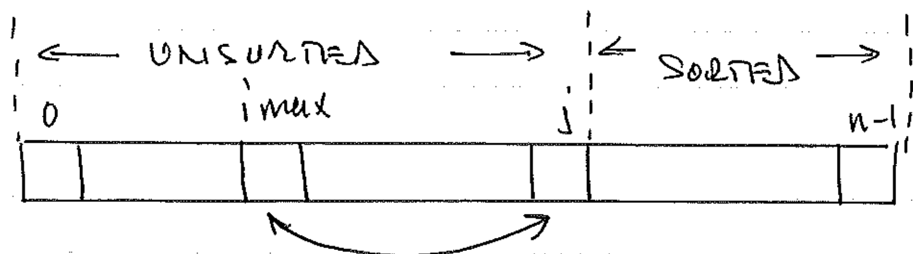
WE TAKE THE BASIC OPERATION TO BE COMPARISON OF ARRAY ELEMENTS.

$$\#\text{Comp} = (n-1) + (n-2) + \dots + 3 + 2 + 1 = \frac{n(n-1)}{2}$$

$\therefore$ RUN TIME IS IN $\Theta(n^2)$

NOTE: THIS IS SAME FOR BEST, WORST, AVERAGE CASES.

```
void SelectionSort(int*A, int n){
    int i, j, temp, imax=0;
    for(j=n-1; j>0; j--){
        for(i=1; i<=j; i++){
            if (A[imax]<A[i])  imax=i;
        }
        temp=A[imax];
        A[imax]=A[j];
        A[j]=temp;
    }
}
```



SWAP MAX ELEMENT IN UNSORTED SECTION WITH RIGHTMOST ELEMENT IN UNSORTED SECTION.

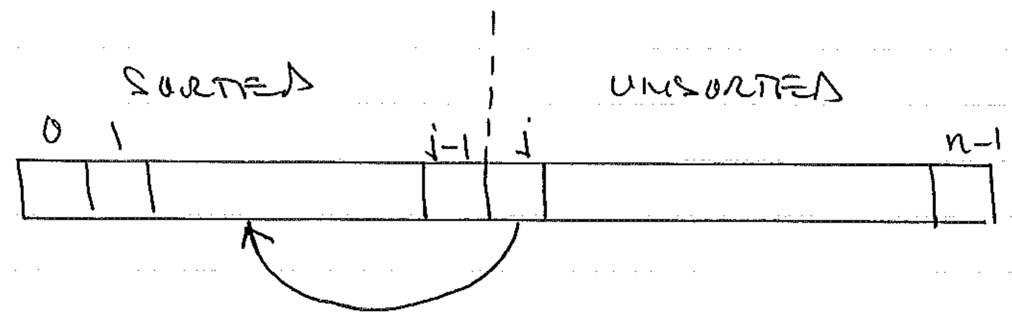
$$\#\text{comp} = (n-1)+(n-2)+\dots+3+2+1 = \frac{n(n-1)}{2}$$

$$= \Theta(n^2) \quad (\text{BEST, WORST, AVERAGE})$$

NOTE THAT IF WE COUNTED ASSIGNMENTS (OR SWAPS) INSTEAD OF COMPARISONS WE GET

	BubbleSort	SelectionSort
#SWAP	$\Theta(n^2)$	$\Theta(n)$

```
void InsertionSort(int * A, int n){
    int i, j, temp;
    for(j=1; j<n; j++){
        temp = A[j]
        i = j-1;
        while(i>=0 && temp<A[i]){
            A[i+1] = A[i]
            i--;
        }
        A[i+1] = temp;
    }
}
```



Insert A[j] INTO THE SORTED SUB ARRAY A[0 ... j-1] AT RIGHT POSITION

BEST CASE # COMP = $n-1$

WORST CASE # COMP = $1+2+3+\cdots+(n-2)+(n-1)$
$= \frac{n(n-1)}{2} = \Theta(n^2)$

AVERAGE CASE # COMP = $\Theta(n^2)$

```
void QuickSort(int* A, p, r) {
    int q;
    if (p<r) {
        q = Partition(A, p, r);
        QuickSort(A, p, q-1);
        QuickSort(A, q+1, r);
    }
}
```

PARTITION RE-ARRANGES THE SUBARRAY $A[p \ldots r]$ AND RETURNS AN INDEX q SUCH THAT

$$A[p \ldots (q-1)] \leq A[q] < A[(q+1) \ldots r]$$

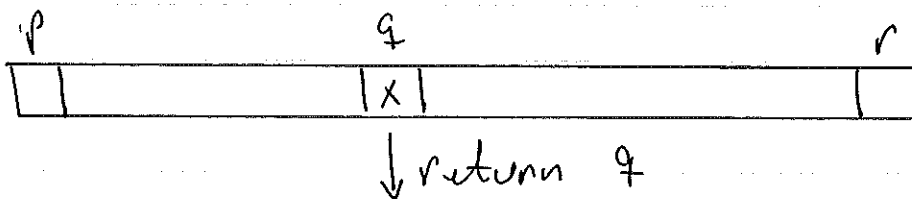
THE ELEMENT $A[q]$ is called THE PIVOT.

```
int Partition(A, p, r) {
    int i, j, x;
    x = A[r];
    i = p-1;
    for (j = p; j<r; j++) {
        if (A[j] [illegible] x) {
            i++;
            swap(A, i, j);
        }
    }
    swap(A, i+1, r);
    return(i+1);
}
```

WE USE A HELPER FUNCTION SWAP WHICH EXCHANGES ARRAY ELEMENTS.

```
void swap(int* A, int i, int j){
    int temp;
    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

PARTITION PICKS AN ELEMENT IN A[p...r] TO BE THE SO-CALLED PIVOT. IN OUR CODE THE PIVOT IS $x = A[r]$. (IN THE BOOK IT A[p].) PARTITION THEN DIVIDES THE SUBARRAY A[p..r] INTO TWO SECTIONS: THOSE ELEMENTS LESS THAN OR EQUAL TO x AND THOSE WHICH ARE GREATER THAN x. IT THEN SWINGS THE PIVOT INTO A POSITION BETWEEN THE TWO SECTIONS, THEN RETURNS THE INDEX OF THE PIVOT.



THUS $A[p \ldots q-1] \leq A[q] < A[q+1 \ldots r]$ UPON RETURN OF PARTITION. ALL THAT IS LEFT IS TO SORT SUBARRAYS $A[p \ldots q-1]$ AND $A[q+1 \ldots r]$ RECURSIVELY, WHICH QUICKSORT DOES.