۱- Big O :

این نماد یک حد بالا برای تابع را مشخص می‌کند. اگر $f(n) = O(g(n))$ بدین معناست که اعداد ثابت مثبت c, n_0 وجود دارند به شرطی که در $n \geq n_0$ داریم $0 \leq f(n) \leq cg(n)$.

۲- Big Omega :

این نماد یک حد پایین برای تابع را مشخص می‌کند. اگر $f(n) = \Omega(g(n))$ بدین معناست اعداد ثابت مثبت c, n_0 وجود دارند به شرطی که در $n \geq n_0$ داریم $f(n) \geq cg(n) \geq 0$.

۳- Big Theta :

این نماد یک حد دوطرفه برای تابع ایجاد می‌کند. اگر $f(n) = \Theta(g(n))$ بدین معناست اعداد ثابت مثبت n_0, c_1, c_2 وجود دارند به شرطی که در $n \geq n_0$ داریم $c_2 g(n) \geq f(n) \geq c_1 g(n) \geq 0$.

۴- Little o :

این نماد یک حد بالا برای تابع را مشخص می‌کند. اگر $f(n) = o(g(n))$ بدین معناست که به ازای هر عدد ثابت مثبت c، عدد ثابت مثبت n_0 وجود دارد به شرطی که در $n \geq n_0$ داریم $cg(n) \geq f(n) \geq 0$.

۵- Little omega :

این نماد یک حد پایین برای تابع را مشخص می‌کند. اگر $f(n) = \omega(g(n))$ باشد بدین معناست که به ازای هر ثابت مثبت c، عدد ثابت مثبت n_0 وجود دارد به شرطی که در $n \geq n_0$ داریم $f(n) > cg(n) \geq 0$.

② فرض کنیم هزینه مرحله جمع برابر c باشد، آنگاه هزینه مرحله ضرب $10c$ باشد.

هزینه برای الگوریتم اول:

$$(1500n)c + (n^2+10n)10c = (1500c)n + (1000c)n + (10c)n^2$$

$$= (2500c)n + (10c)n^2$$

هزینه برای الگوریتم دوم:

$$(500n)c + (5n^2)10c = (500c)n + (50c)n^2$$

(a

برای اینکه سرعت الگوریتم اول از الگوریتم دوم بیشتر باشد، باید هزینه آن کمتر شود یعنی:

$$(10c)n^2 + (2500c)n < (50c)n^2 + (500c)n \xrightarrow{\times \frac{1}{10c}}$$

$$n^2 + 250n < 5n^2 + 50n \longrightarrow 0 < 4n^2 - 200n$$

$$\rightarrow 0 < 4n(n-50)$$

$$\begin{cases} n>50 : + \checkmark \\ 0<n<50 : - \times \\ n<0 : + \times \end{cases}$$

یعنی: از ای $n>50$ الگوریتم اول از الگوریتم دوم سریعتر است

(b

اردر زمانی هردو الگوریتم از مرتبه $O(n^2)$ است.

۹) ساده‌ترین راه‌حل استفاده از الگوریتم حلی است که در آن متغیری به نام max داشته باشیم که در شروع برابر با خانه با اندیس صفر آرایه است و با پیمایش بررسی آرایه و مقایسه با مقدار درخانه بعدی مقدار آن خانه از max بیشتر باشد آن خانه جایگزین مقدار max می‌شود. شبه‌کد:

	cost	times
Max_Finder (A)		
max ← A[0]	c_1	1
for (i=1 ; i<n ; i++)	c_2	n
if (A[i] > max)	c_3	n-1
max ← A[i]	c_4	n-1

اردر زمانی بدترین حالت برابر است با

$$T(n) = c_1 \times 1 + c_2 \times n + c_3 \times (n-1) + c_4 \times (n-1)$$

$$= (c_2 + c_3 + c_4)n + c_1 - c_3 - c_4 = O(n)$$

b) راه حل برای این مسئله م همانند قسمت قبل است که یکبار بررسی این آرایه iterate کنیم و اینبار بجای فقط پیداکردن ماکسیمم، چهار مقدار بزرگترین عدد صحت، دومین بزرگترین عدد صحت، کوچکترین عدد صحتی، دومین کوچکترین عدد صحتی را پیدا می‌کنیم و حاصلضرب دو عدد صحت و دو عدد صحتی را می‌سنجیم و حاصل بزرگتر را به عنوان جواب برمی‌گردانیم. شبه‌کد در بدترین حالت:

	Cost	times
Max_Product_Finder (A)		
Max_Pos , Sec_Max_Pos, Min_Neg, Sec_Min_Neg = 0	C_1	1
for (int i=0 ; i < n ; i++)	C_2	n+1
if (A[i] > Max_Pos) → if(A[i] ≥ 0)	C_3	n/2
Sec_Max_Pos = Max_Pos	C_4	n/2
Max_Pos = A[i]	C_5	n/2
else if (A[i] > Sec_Max_Pos)	C_6	0
Sec_Max_Pos = A[i]	C_7	0
else	C_8	n/2
if (A[i] < Min_Neg)	C_9	n/2
Sec_Min_Neg = Min_Neg	C_{10}	n/2
Min_Neg = A[i]	C_{11}	n/2
else if (A[i] < Sec_Min_Neg)	C_{12}	0
Sec_Min_Neg = A[i]	C_{13}	0
Product_Pos = Max_Pos * Sec_Max_Pos	C_{14}	1
Product_Neg = Min_Neg * Sec_Min_Neg	C_{15}	1
return MAX (Product_Pos , Product_Neg)	C_{16}	1

اردر زمانی =>

$$T(n) = \left(C_2 + \frac{C_3}{2} + \frac{C_4}{2} + \frac{C_5}{2} + \frac{C_8}{2} + \frac{C_9}{2} + \frac{C_{10}}{2} + \frac{C_{11}}{2} - \frac{C_{12}}{2}\right) n$$

$$+ (C_1 + C_2 + C_{14} + C_{15} + C_{16}) = O(n)$$

۱۲

a)

۱- $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = O(n)$

۲- $\sum_{i=1}^{n} \sqrt{i} \simeq \frac{2}{3}(n^{3/2} - 1) = O(n^{3/2})$

۳- $\sum_{i=1}^{n} i^{-1} = O(\ln(n))$

۴- $\sum_{i=1}^{n} \log(i) = \log(1) + \log(2) + \cdots + \log(n) = \log(1 \times 2 \times 3 \times \cdots \times n) = \log n! = O(n \log(n))$

b)

ترتیب سریع رشد: $O(n^{3/2}) > O(n \log(n)) > O(n) > O(\ln(n))$

تنها جایی است در حلقه while داخلی در حمان در insertion sort که شمارنده داشته باشیم که تعداد جابجایی یا inversion ها را بشمرد.

```
Insertion_Sort(A)                    → inversions ← 0
"
for (int i=1; i<n; i++)
    key = A[i]
    j = i-1
    while (j≥0 & A[j] > key)
        A[j+1] = A[j]
        j--
        inversions++
    A[j+1] = key

return inversions
```