CQ 1:

How many *operators* will be executed?

n sum
1 0
3 1
5 4
7 9.

```
sum = 0;      // 1
n = 1;        // 1
while (n < 10 && sum < 5) { // 3*4 times
  sum = sum + n; // 2 * 3 times.
  n = n + 2; // 2 * 3 times.
}
```

1 + 1 + 12 + 6 + 6 = 26

A <= 20

B 21..25

C * 26..30

D 31..35

E >= 36

solutions.

Data size

a[i] = *(a+i)   2 operations
② ①

What is the number of operations executed for this implementation?

```
int sum_array(const int a[], int len) {      indices are 0,1,2,....,n-1.
                                   n
  int sum = 0;     //1
  int i = 0;      //1
             n
  while (i < len) {     //1*(n+1) times   i=0,1,...,n-1,n.
    sum = sum + a[i];   //4*n times
    i = i + 1;     //2*n times
  }
  return sum;          1+1+(len+1)+4len+2len=7len+3
}
```

The running time **depends on the length** of the array.

If there are n items in the array, it requires $7n + 3$ operations.

> We are always interested in the running time *with respect to* the **size of the data**.

solutions

Bart just wants to count the total number of odd numbers in the entire array.

```
bool bart(const int a[], int len, int e, int o) {
  int odd_count = 0;   int i=0;    // 2
  for (~~int i = 0~~; i < len; ~~i = i + 1~~) { // 1*(n+1)
    odd_count = odd_count + (a[i] % 2); // 5*n
  }  i=i+1;  // 2*n
  return (odd_count >= o) && (len - odd_count >= e); //4
}
```

$2 + (n+1) + 5n + 2n + 4 = 8n + 7$

If there are n elements in the array, $T(n) = 8n + 7$.

Go thru array once regardless of its content.

A constant # of operations for each element.

Remember, you are not expected to calculate this precisely.

Solutions

Homer is lazy, and he doesn't want to check all of the elements in the array if he doesn't have to.

```
bool homer(const int a[], int len, int e, int o) {
  // only loop while it's still possible
  while (len > 0 && e + o <= len) {   // only start if total # of required #s
    if (a[len - 1] % 2 == 0) {  // even case:   is no more than # of
      if (e > 0) {                               elements in a.
        e = e - 1;               // only decrement e if e > 0
      }
    } else if (o > 0) {     // odd case (if we have more odd #'s to find,
      o = o - 1;                         decrement o.)
    }
    if (e == 0 && o == 0) {  // we have found enough even & odd #'s,
      return true;
    }
    len = len - 1;  // go to prev element.
  }
  return false;
}
```

Start from the back of the array.

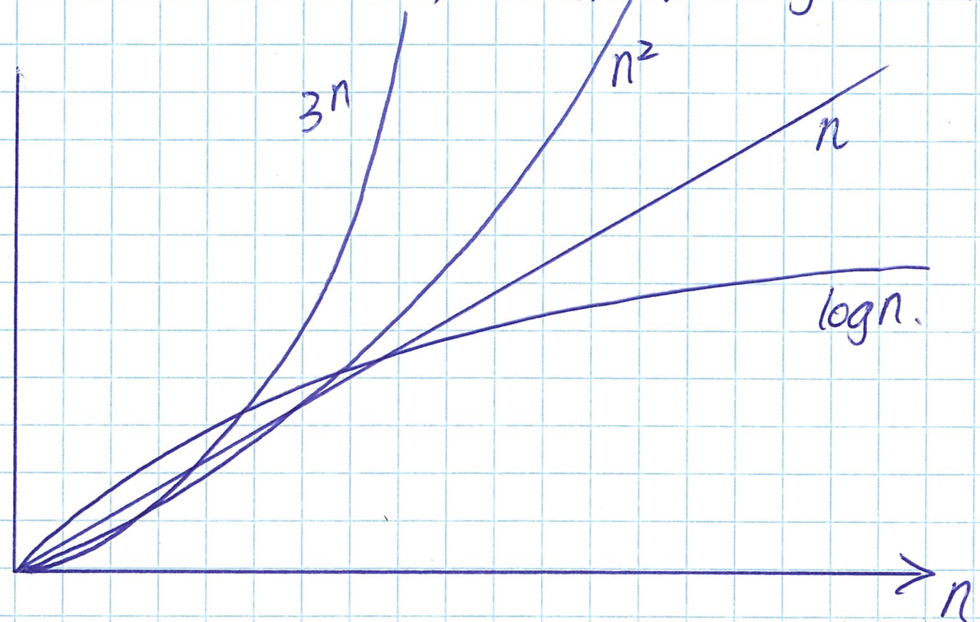
Solutions

exponential polynomial logarithmic

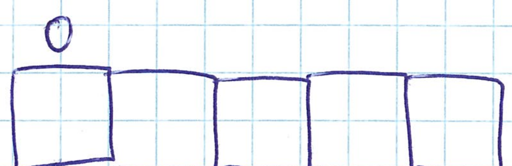
$$3^n > \underbrace{2n^3}_{\text{cubic}} \ \underbrace{3n^2}_{\text{quadratic}} > 4\underbrace{n\log n} > \underbrace{5n^1}_{\text{linear}} > 6\underbrace{\log n}. > \underbrace{8n^0}_{\text{constant}}.$$

Dominant term: (Slides 17 and 18 Section 8)

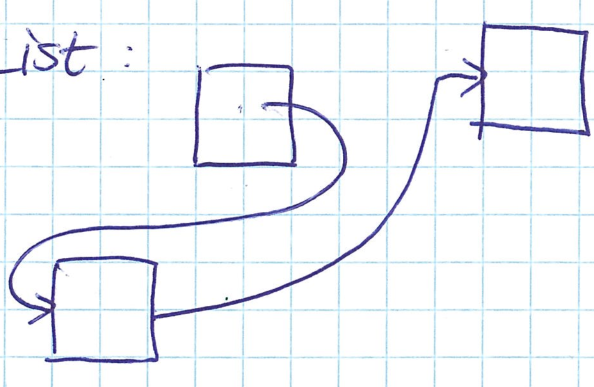
- ignore constant coefficients.
- when n is large, which term becomes greatest?
- as n increases, which term grows fastest?



Arrays:



List:



Section 8 Slide 32

Why is $\sum_{i=1}^{n} O(i) = O(n^2)$?

$$
\begin{array}{l}
1 + 2 + 3 + \cdots + (n-2) + (n-1) + n = \frac{(n+1)n}{2} \\
n + (n-1) + (n-2) + \cdots + 3 + 2 + 1
\end{array}
$$

$$(n+1) + (n+1) + (n+1) + \cdots + (n+1) + (n+1) + (n+1)$$

$$= (n+1)n$$

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

Section 8 Slide 34.

How many times can we divide k by 10 before k becomes 0?

If we can divide k by 10 for x times,

it must be that $n \approx 10^x$.

so $x \approx \log_{10} k$