**Complexity**

* T(n) means the time it takes to run a task
* Benchmark: a = b 🡪 T(n) = 1

sum = 0;

for (int = 0; i < n; i++) 🡪 T(n) = 2 + 3\*n

sum += i;

* Linear Search (3 cases)

**Big O notation**

* We are only interested in the highest order term (the only term that finally dominates)
* Intuitively: if we have 4n3 + 5n5 + n + 1 🡪 choose n5
* When we talk about O, we are talking about **tight bound** **🡪 Worst case**
* Therefore, O(nk) is a family of function:
  + O(n) includes an, an + b
  + O(n2) includes an2, an2 + bn + c, etc.
  + **NOTE** that O(log(n)) is equivalent to O(log2(n)), O(log9(n)), … (remove constant)

sum = 0;

for (int = 0; i < n; i++) 🡪 T(n) = 2 + 3\*n 🡪 O(n)

sum += i;

T(n) = 4 🡪 O(1)

Example: **Matrix multiplication**

int i, j, k;

for (int i =0; i < N; i++)

for (int j = 0; j < N; j++) 🡪 O(n3)

for (int k = 0; k < N; k++)

c[i][j] += a[i][k] \* b[k][j];

Example: **Facebook Homepage**

foreach (friend in friendList) // assume n friends

foreach (update of friend) // assume m recent updates

display update;

🡪 O(nxm) // would be O(n) if m is constant

Example: **Insertion sort**

for (i = 0; i < n; i++)

for (j = i; j > 0 && a[j] < a[j-1]; j--)

swap (a[j], a[j-1]);

Best case: O(n) // still have to go through all elements (never swapped)

Worst case: O(n2) // Worst case: T(n)≈ C(n + (n-1) + (n-2) + .. +1) = C\*n(n+1)/2 = O(n2)

Average case: O(n2)

Text, application

Description automatically generatedExample: **Quick sort**

T(n) = TselectAndShuffle(n) + T(k) + T(n - k - 1)

🡪 T(n) ≈ TselectAndShuffle(n) + T(k) + T(n - k - 1)

≈ C×n + T(k) + T(n-k-1)

Text

Description automatically generated

T(n) ≈ C × n

Worst case: k = 0, i.e., pivotIndex == begin

Diagram

Description automatically generatedT(n) ≈ C×n + T(n-1)

≈ C×(n + n-1) + T(n-2)

…

≈ C×(n + n-1 + n-2 + … 1)

= C×n×(n+1)/2 = O(n2)

Best/average case: O( n\*log(n) )

Worst case: O(n2) 🡪 pivot far left/right

Example: **Binary Search on Sorted Array**

bool search (int first, int last, int key, int & index)

{

if (first > last) return false;

int mid = (first + last) /2 ;

if (key == a[mid]) {

index = mid;

return true;

}

if (key < a[mid])

return search (first, mid-1, key, index);

else

return search(mid+1, last, key, index);

}

Problem size: n, n/2, n/4,… 1 🡪 T(n) = C + T(n/2) = C + (C + T(n/4)).. 🡪 O(log(n))

Example: **Search on Binary Search Tree**

Worst case: A binary search tree completely unbalanced (only left/right subtree) 🡪 O(n)

Best case: O(1)

Average case: Every time, we are eliminating half of the elements 🡪 O(log(n))

Diagram

Description automatically generated

**Cache and Memory**

However, there are other concerns:

* CPU: speed in ns
* Latency of **CPU Cache**: in ns
* Memory: 100s ns

🡪 Sometimes, the constant C in algorithm is sometimes not important, that we can make it as large as possible to fit in the CPU cache! 🡪 faster

e.g. Matrix multiplication:

- Naïve version: a bunch of for loops

- When the matrix gets big, we may not fit the size in the Cache 🡪 If we really want to speed up matrix multiplication, we do multiplication in small regions of the matrix

🡪 Increasing big C 🡪 But gets hundreds of times of time optimization.

e.g. Sort:

Let's say we want to sort for an array. When array gets large, the array cannot fit in the cache.

🡪 If we have some sorting algorithm that works on small partions of the array (sub-arrays) 🡪 fit into cache

🡪 **Quick sort is actually faster than Merge sort, even though they have same complexity!**

Table

Description automatically generated

**\*\*\* How to calculate complexity**

T(n) = n + 2 T(n/2)

= n + 2 (n/2 + 2 T(n/4)) = n + n + 4 T(n/4)

= n + n + …. + a T(n/a)

----------------- -------------- Stop when n = a

log2(a) times n T(1)

🡪 n\*log2(n) + n 🡪 nlog(n)