Iterative Algorithm Analysis & Asymptotic Notations

- Iterative Algorithms and their analysis

- Asymptotic Notations
  - Big O, Θ (theta), Ω (omega) Notations

- Review of Discrete Math
  - Summations
  - Logarithms

Example I:
Finding the sum of an array of numbers

```
int Sum(int A[], int N) {
  int sum = 0;

  for  (i=0; i < N; i++){
    sum += A[i];
  } //end-for

  return sum;
} //end-Sum
```

- How many steps does this algorithm take to finish?
  - We define a step to be a unit of work that can be executed in constant amount of time in a machine.

Example I:
Finding the sum of an array of numbers

Times Executed

```
int Sum(int A[], int N) {
  int sum = 0;                    1

  for (i=0; i < N; i++){          N
    sum += A[i];                  N
  } //end-for

  return sum;                     1
} //end-Sum
```

Total: 1 + N + N + 1 = 2N + 2

- Running Time: $T(N) = 2N+2$
  - N is the input size (number of ints) to add

Example II:
Searching a key in an array of numbers

```
int LinearSearch(int A[], int N,
                 int key) {
  int i = 0;

  while (i < N){
    if (A[i] == key) break;
    i++;
  } //end-while

  if (i < N) return i;
  else return -1;
} //end-LinearSearch
```

Times Executed

1

1<=L<=N

1<=L<=N

0<=L<=N

1

1

Total: 1+3*L+1 = 3L+2

Example II:
Searching a key in an array of numbers

- What's the best case?
  - Loop iterates just once =>T(n) = 5
- What's the average (expected) case?
  - Loop iterates N/2 times =>T(n)=3*n/2+2 = 1.5n+2
- What's the worst case?
  - Loop iterates N times =>T(n) = 3n+2

Worst Case Analysis of Algorithms

- We will only look at WORST CASE running time of an algorithm. Why?
  - Worst case is an upper bound on the running time. It gives us a guarantee that the algorithm will never take any longer
  - For some algorithms, the worst case happens fairly often. As in this search example, the searched item is typically not in the array, so the loop will iterate N times
  - The "average case" is often roughly as bad as the "worst case". In our search algorithm, both the average case and the worst case are linear functions of the input size "n"

Example III: Nested for loops

```
for (i=1; i<=N; i++){
    for (j=1; j<=N; j++){
        println("Foo\n");
    } //end-for-inner
} //end-for-outer
```

- How many times is the printf statement executed?
    - Or how many Foo will you see on the screen?

$$T(N)=\sum_{i=1}^{N}\sum_{j=1}^{N}1=\sum_{i=1}^{N}N=N*N=N^2$$

Example IV: Matrix Multiplication

```
/* Two dimensional arrays A, B, C. Compute C = A*B */
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        C[i][j] = 0;
        for (int k=0; k<N; k++){
          C[i][j] += A[i][k]*B[k][j];
        } //end-for-innermost
    } //end-for-inner
} //end-for-outer
```

$$T(N) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left(1 + \sum_{k=0}^{N-1} 1\right) = N^3 + N^2$$

Example V: Binary Search

- Problem: You are given a sorted array of integers, and you are searching for a key
  - Linear Search – $T(n) = 3n+2$ (Worst case)
  - Can we do better?
  - E.g. Search for 55 in the sorted array below

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	8	10	11	20	50	55	60	65	70	72	90	91	94	96	99

Example V: Binary Search

- Since the array is sorted, we can reduce our search space in half by comparing the target key with the key contained in the middle of the array and continue this way
- Example: Let's search for 55

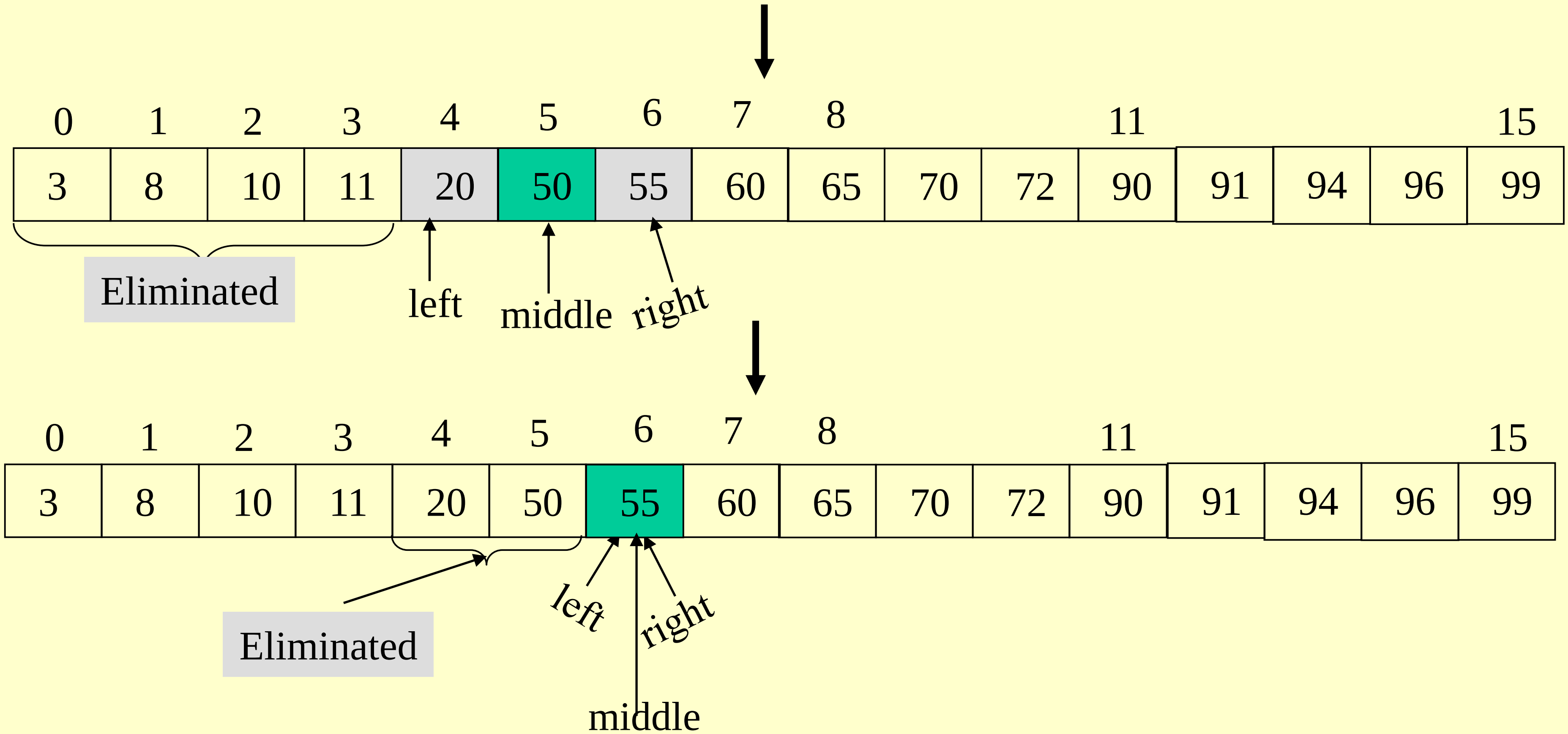
0	1	2	3	4			7	8			11				15
3	8	10	11	20	50	55	60	65	70	72	90	91	94	96	99
left							middle								right

$$middle = \frac{(left + right)}{2}$$

0	1	2	3	4		6	7	8			11				15
3	8	10	11	20	50	55	60	65	70	72	90	91	94	96	99
left			middle			right									

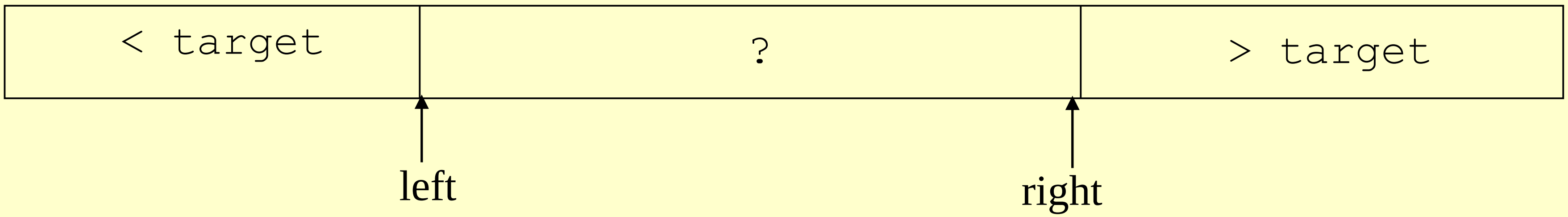
Eliminated

Binary Search (continued)



- Now we found 55→ Successful search
- Had we searched for 57, we would have terminated at the next step unsuccessfully

Binary Search (continued)



- At any step during a search for "target", we have restricted our search space to those keys between "left" and "right".

- Any key to the left of "left" is smaller than "target" and is thus eliminated from the search space

- Any key to the right of "right" is greater than "target" and is thus eliminated from the search space

Binary Search - Algorithm

```
// Return the index of the array containing the key or –1 if key not found
int BinarySearch(int A[], int N, int key){
    left = 0;
    right = N-1;

    while (left <= right){
        int middle = (left+right)/2;            // Index of the key to test against
        if (A[middle] == key) return middle;   // Key found. Return the index
        else if (key < A[middle]) right = middle – 1;   // Eliminate the right side
        else left = middle+1;                          // Eliminate the left side
    } //end-while

    return –1;   // Key not found
} //end-BinarySearch
```

- Worst case running time: $T(n) = \log_2 N$. Why?

Binary Search - Algorithm

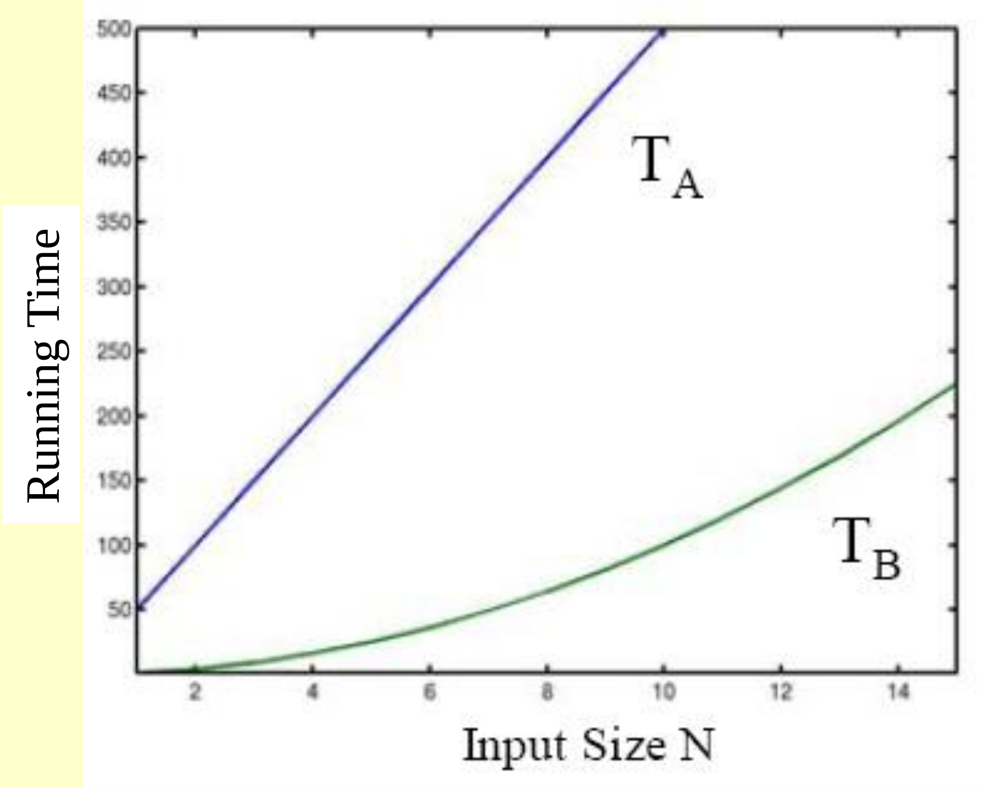
How do we obtain a running time depending on logN?

- After first iteration: N/2 items remaining
- After 2nd iteration: (N/2)/2 = N/4 remaining
- After Kth iteration: $\mathbf{N/2^K}$ remaining
- Worst case: Last iteration occurs when $\mathbf{N/2^K} \geq 1$
- $\mathbf{2^K} \leq N$
- take log of both sides
- Number of iterations is $K \leq \log N$

Motivation for Asymptotic Notation

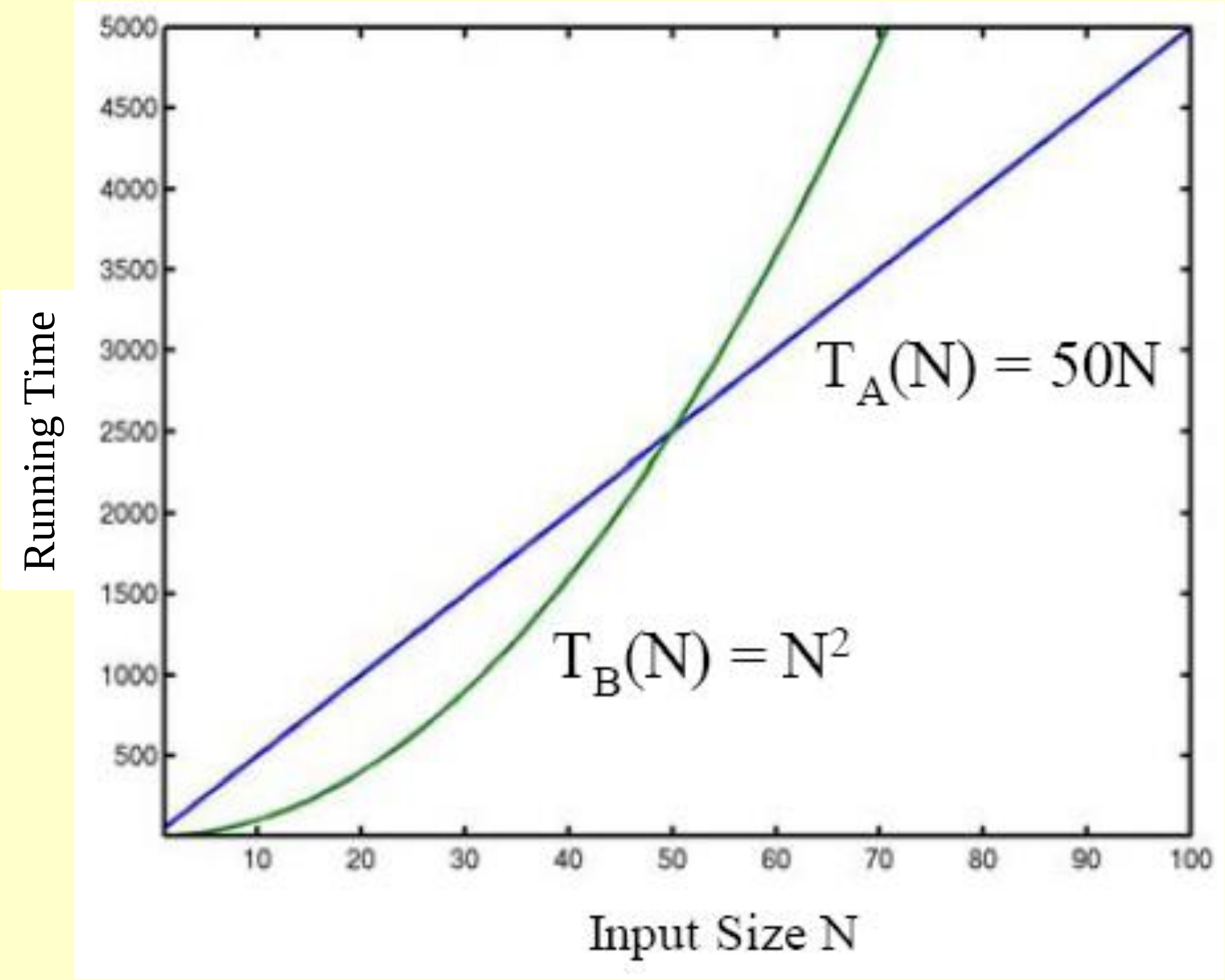
- Suppose you are given two algorithms A and B for solving a problem
- Here is the running time $T_A(N)$ and $T_B(N)$ of A and B as a function of input size N:

- Which algorithm would you choose?



Motivation for Asymptotic Notation

- For large N, the running time of A and B is:



- Which algorithm would you choose now?

Motivation for Asymptotic Notation

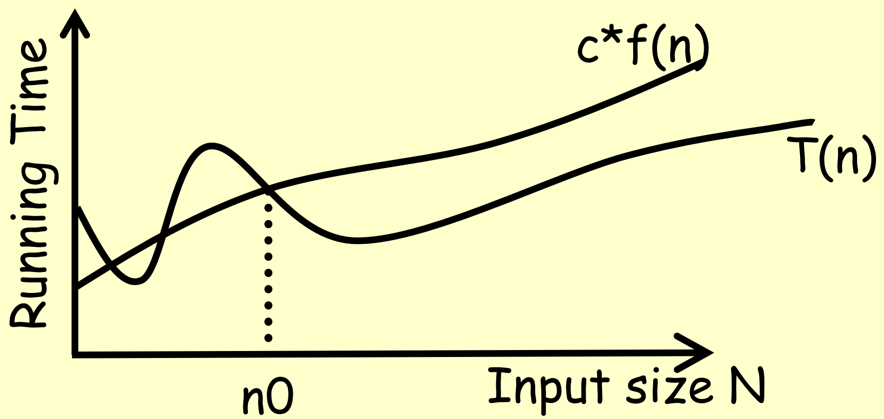
- In general, what really matters is the "asymptotic" performance as $N \to \infty$, regardless of what happens for small input sizes N.

- Performance for small input sizes may matter in practice, if you are sure that small N will be common
  - This is usually not the case for most applications

- Given functions T1(N) and T2(N) that define the running times of two algorithms, we need a way to decide which one is better (i.e. asymptotically smaller)
  - Asymptotic notations
  - Big-Oh, Ω, Θ notations

Big-Oh Notation: Asymptotic Upper Bound

- T(n) = O(f(n)) [T(n) is big-Oh of f(n) or order of f(n)]
  - If there are positive constants c & n0 such that
    T(n) <= c*f(n) for all n >= n0



- Example: T(n) = 50n is O(n). Why?
  - Choose c=50, n0=1. Then 50n <= 50n for all n>=1
  - many other choices work too!

Another example: https://xlinux.nist.gov/dads/HTML/bigOnotation.html

Big-Oh Notation: Asymptotic Upper Bound

- $T(n) = O(f(n))$
  - If there are positive constants c & n0 such that $T(n) \le c \cdot f(n)$ for all $n \ge n0$

- Example: $T(n) = 2n+5$ is $O(n)$ why?
  - We want $T(n) = 2n+5 \le c \cdot n$ for all $n \ge n0$
  - $2n+5 \le 2n+5n \le 7n$ for all $n \ge 1$
    - c = 7, no = 1
  - $2n+5 \le 3n$ for all $n \ge 5$
    - c = 3, no=5
  - Many other c & no values would work too

Big-Oh Notation: Asymptotic Upper Bound

- $T(n) = O(f(n))$
  - If there are positive constants c & n0 such that $T(n) <= c*f(n)$ for all $n >= n0$

- Example: $T(n) = 2n+5$ is $O(n^2)$ why?
  - We want $T(n) = 2n+5 <= c*n^2$ for all $n>=n0$
  - $2n+5 <= 1*n^2$ for all $n>=4$
    - c = 1, no = 4
  - $2n+5 <= 2*n^2$ for all $n>=3$
    - c = 2, no = 3
  - Many other c & no values would work too

Big-Oh Notation: Asymptotic Upper Bound

- T(n) = O(f(n))
  - If there are positive constants c & n0 such that T(n) <= c*f(n) for all n >= n0

- Example: T(n) = n(n+1)/2 is O(?)

  - T(n) = $n^2/2 + n/2$ is O(N^2). Why?

  - $n^2/2 + n/2 <= n^2/2 + n^2/2 <= n^2$ for all n >=1
  - So, T(n)=n*(n+1)/2 <= 1* n^2 for all n >=1
    - c=1, no=1

Common Functions we will encounter

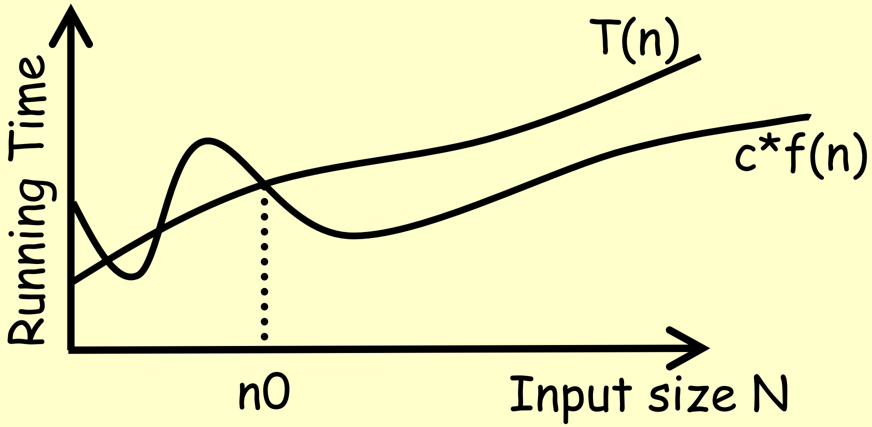
Increasing cost

Name	Big-Oh	Comment
Constant	$O(1)$	Can't beat it!
Log log	$O(\log\log N)$	Extrapolation search
Logarithmic	$O(\log N)$	Typical time for good searching algorithms
Linear	$O(N)$	This is about the fastest that an algorithm can run given that we need $O(n)$ just to read the input
N logN	$O(N\log N)$	Most sorting algorithms
Quadratic	$O(N^2)$	Acceptable when the data size is small ($N<1000$)
Cubic	$O(N^3)$	Acceptable when the data size is small ($N<1000$)
Exponential	$O(2^N)$	Only good for really small input sizes ($n<=20$)

Polynomial time

Ω Notation: Asymptotic Lower Bound

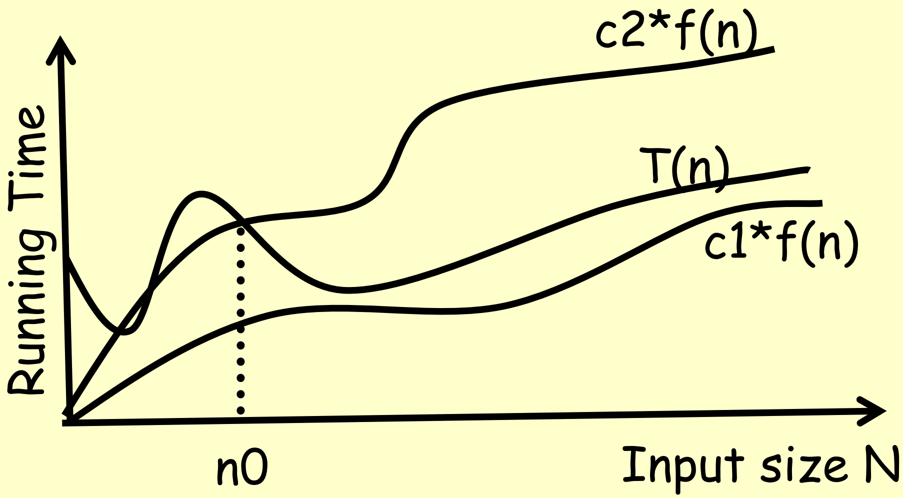
- $T(n) = \Omega(f(n))$
  - If there are positive constants c & n0 such that $T(n) >= c*f(n)$ for all $n >= n0$



- Example: $T(n) = 2n + 5$ is $\Omega(n)$. Why?
  - $2n+5 >= 2n$, for all $n >= 1$
- $T(n) = 5*n^2 - 3*n$ is $\Omega(n^2)$. Why?
  - $5*n^2 - 3*n >= 4*n^2$, for all $n >= 4$

Θ Notation: Asymptotic Tight Bound

- $T(n) = \Theta(f(n))$
  - If there are positive constants c1, c2 & n0 such that $c1*f(n) <= T(n) <= c2*f(n)$ for all $n >= n0$



  - Example: $T(n) = 2n + 5$ is $\Theta(n)$. Why?

    $2n <= 2n+5 <= 3n$, for all $n >= 5$

  - $T(n) = 5*n^2 - 3*n$ is $\Theta(n^2)$. Why?
    - $4*n^2 <= 5*n^2 - 3*n <= 5*n^2$, for all $n >= 4$

Some Math

S(N) = 1 + 2 + 3 + 4 + … N = $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$

Sum of Squares: $\sum_{i=1}^{N} i^2 = \frac{N*(N+1)*(2n+1)}{6} \approx \frac{N^3}{3}$

Geometric Series: $\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$ A > 1

$$\sum_{i=0}^{N} A^i = \frac{1-A^{N+1}}{1-A} = \Theta(1)$$ A < 1

Some More Math

Linear Geometric Series:

$$\sum_{i=0}^{n} ix^i = x + 2x^2 + 3x^3 + \ldots + nx^n = \frac{(n-1)x^{(n+1)} - nx^n + x}{(x-1)^2}$$

Harmonic Series:

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = (\ln n) + O(1)$$

Logs:

$$\log A^B = B * \log A$$

$$\log(A * B) = \log A + \log B$$

$$\log(\frac{A}{B}) = \log A - \log B$$

More on Summations

- Summations with general bounds:

$$\sum_{i=a}^{b} f(i) = \sum_{i=0}^{b} f(i) - \sum_{i=0}^{a-1} f(i)$$

- Linearity of Summations:

$$\sum_{i=1}^{n} (4i^2 - 6i) = 4\sum_{i=1}^{n} i^2 - 6\sum_{i=1}^{n} i$$