



Contents

- What is an algorithm
- What is complexity analysis
- Example of different complexities:
 - maximum subsequence sum problem
- Big-O notation
- Analysis of Example Codes



Algorithm and Complexity Analysis

- An algorithm is a clearly specified set of simple instructions to be followed to solve a problem.
- Analysis of an algorithm gives insight into how long the program runs and how much memory it uses
 - time complexity - our main concern
 - space complexity



Algorithm Efficiency

- Introducing Maximum Subsequence Sum Problem
- There can be 4 different algorithms to solve the above problem.



Maximum Subsequence Sum Problem

- Given (possibly negative) integers A_1, A_2, \dots, A_N , find the **maximum value of** $\sum_{k=i}^j A_k$
- Example:
 - For input $-2, 11, -4, 13, -5, -2$, the answer is 20



Demo Time!

- The next several slides compare 4 different algorithms for the max subsequence sum problem.
- At this stage, it is ok if you don't fully understand the implementation details. We will revisit.
- This is mainly to give you a sense of the different time complexity among different algorithms.



Algorithm 1: Brute-force method

- Given n integers A_1, A_2, \dots, A_N , what are the maximum number of different sums you can form, assuming each number appears only once in each sum?
- For example: 1, 2, 3, 4
 - The possible sums include:
 - 1, 1+2, 1+2+3, 1+2+3+4
 - 2, 2+3, 2+3+4
 - 3, 3+4
 - 4



Algorithm 1: Brute-force method:

simply computes all possible sums and find the largest one.

```
long maxSubSum1(const vector<long> & a)
{
    long maxSum = 0;
    for (int i = 0; i < a.size(); i++)
        for(int j = i; j < a.size(); j++)
        {
            long thisSum = 0;
            for (int k = i; k <=j; k++)
                thisSum += a[ k ];
            if (thisSum > maxSum)
                maxSum = thisSum;
        }
    return maxSum;
}
```



Algorithm 1: Brute-force method

- Algorithm 1 simply computes all possible sums and find the largest one.
- An $O(N^3)$ solution (Cubic complexity)
 - The Big-O notation indicates the upperbound of running time as the function of input size. Will be defined later.



Algorithm 2: A Bit Clever Algorithm

- By noting

$$\sum_{k=i}^j A_k = A_j + \sum_{k=i}^{j-1} A_k$$

- We can reuse partial computation in previous steps



Algorithm 2: A Bit Clever Algorithm

```
long maxSubSum2(const vector<long> & a)
{
    long maxSum = 0;
    for (int i = 0; i < a.size(); i++)
    {
        long thisSum = 0;
        for ( int j = i; j < a.size(); j++)
        {
            thisSum +=a[ j ];
            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }
    return maxSum;
}
```



Algorithm 2: A Bit Clever Algorithm

- An $O(N^2)$ algorithm (Quadratic complexity. Better than Cubic.)



Algorithm 3: Divide and Conquer

- In second half: 8

First Half				Second Half			
4	-3	5	-2	-1	2	6	-2

Algorithm 3: Divide and Conquer

■ Crosses middle: 11

First Half				Second Half			
4	-3	5	-2	-1	2	6	-2

The maximum sum in the 1st half that includes the last element in the 1st half: 4 (A1 to A4); the maximum sum in the 2nd half that includes the 1st element in the 2nd half is 7 (A5 to A7), so the max sum that spans both halves is $4 + 7 = 11$.

So in this case, **the best answer among three** (1st half, 2nd half, cross) **is the one that crosses both halves: 11.**



Algorithm 3: Divide and Conquer

- Divide Part:

- Split the problem into two roughly equal subproblems, which are then solved recursively.

- Conquer Part:

- Patching together the two solutions of the subproblems with small amount of additional work.



Algorithm 3: Divide and Conquer

- Maximum subsequence sum problem
 - Divide the number sequence into two equal parts
 - The maximum subsequence sum can be found in one of three places:
 - Entirely in the left half of the input
 - Entirely in the right half of the input
 - Crosses the middle and in both halves



Algorithm 3: Divide and Conquer


- In first half: 6

First Half				Second Half			
4	-3	5	-2	-1	2	6	-2



Algorithm 3: Divide and Conquer

- An $O(N \log N)$ algorithm (log-linear)
- Detailed code analysis not required for now.



```

int maxSumRec(const vector<int> &a, int left, int right)
{
    if (left == right)
        if (a[left] > 0)
            return a[left];
        else
            return 0;
    int center = (left + right)/2;
    int maxLeftSum = maxSumRec(a, left, center);
    int maxRightSum = maxSumRec(a, center+1, right);

    int maxLeftBorderSum = 0, leftBorderSum = 0;
    for (int i = center; i <= left; i--)
    {
        leftBorderSum += a[i];
        if (leftBorderSum > maxLeftBorderSum)
            maxLeftBorderSum = leftBorderSum;
    }

    int maxRightBorderSum = 0, rightBorderSum = 0;
    for (int j = center + 1; j <= right; j++)
    {
        rightBorderSum += a[j];
        if (rightBorderSum > maxRightBorderSum)
            maxRightBorderSum = rightBorderSum;
    }

    return max3(maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum);
}

```



Driver function

```
■ int maxSubSum3(const vector<int> &a)
■ {
■     return maxSumRec(a, 0, a.size()-1);
■ }
```



Algorithm 4: The most clever one

- An improvement over algorithm 2
- Clever observations:
 - No negative subsequence can possibly be a prefix of the optimal subsequence
 - Example:
 - -2 6 8 -2 won't be the start
 - 4 -5 8 0 ('4 -5' \rightarrow -1, won't be the start)
 - 2 4 -5 8 0



Algorithm 4: The most clever one

```
long maxSubSum4(const vector<long> & a)
{
    long maxSum = 0, thisSum = 0;
    for( int j = 0; j <a.size(); j++)
    {
        thisSum += a[j];
        if(thisSum > maxSum)
            maxSum = thisSum;
        else if (thisSum < 0)
            thisSum = 0;
    }
    return maxSum;
}
```



Algorithm 4: The most clever one

- An $O(N)$ algorithm (linear, best among 4)



Revisit “Algorithm” (Lecture can start from this slide.)

- An algorithm is a clearly specified set of simple instructions to be followed to solve a problem.
- More specifically, an algorithm is
 - A finite steps of operations (comparisons, additions, assignments, etc.)
 - Work on input of data of finite size
 - Produce desired output and terminate within certain time limit.



Revisit “Analysis of Algorithms”

- Analysis of an algorithm gives insight into how long the program runs and how much memory it uses
 - time complexity
 - space complexity
- Why useful?
 - Problems are often of a large scale. Imagine web server, social media network. Time matters! (Usually a big concern)



Algorithm Analysis

■ Correctness

- Testing
- Proofs of correctness

■ Efficiency

- How to define?
- *Asymptotic complexity* - how running times scales as **function of size of input**

How long to execute:

n	$t(n)$	$t(\log_2 n)$	$t(n \log_2 n)$	$t(n^2)$	$t(2^n)$
10	$0.01 \mu s$	$0.003 \mu s$	$0.033 \mu s$	$0.1 \mu s$	$1 \mu s$
20	$0.02 \mu s$	$0.004 \mu s$	$0.086 \mu s$	$0.4 \mu s$	$1 ms$
30	$0.03 \mu s$	$0.005 \mu s$	$0.147 \mu s$	$0.9 \mu s$	$1 s$
40	$0.04 \mu s$	$0.005 \mu s$	$0.213 \mu s$	$1.6 \mu s$	18.3min
50	$0.05 \mu s$	$0.006 \mu s$	$0.282 \mu s$	$2.5 \mu s$	13 days
100	$0.10 \mu s$	$0.007 \mu s$	$0.664 \mu s$	$10 \mu s$	4×10^{13} years
1,000	$1.00 \mu s$	$0.010 \mu s$	$9.966 \mu s$	$1 ms$	
10,000	$10 \mu s$	$0.013 \mu s$	$130 \mu s$	$100 ms$	
100,000	$0.10 ms$	$0.017 \mu s$	$1.67 ms$	$10 s$	
1,000,000	$1 ms$	$0.020 \mu s$	$19.93 ms$	$16.7 min$	
10,000,000	$0.01 s$	$0.023 \mu s$	$0.23 s$	$1.16 days$	

n is a number of items to process, and $t(x)$ is how much time it would take to process them all with an algorithm of complexity x , using a computer that does 1 *billion* operations per second.

n up to 8

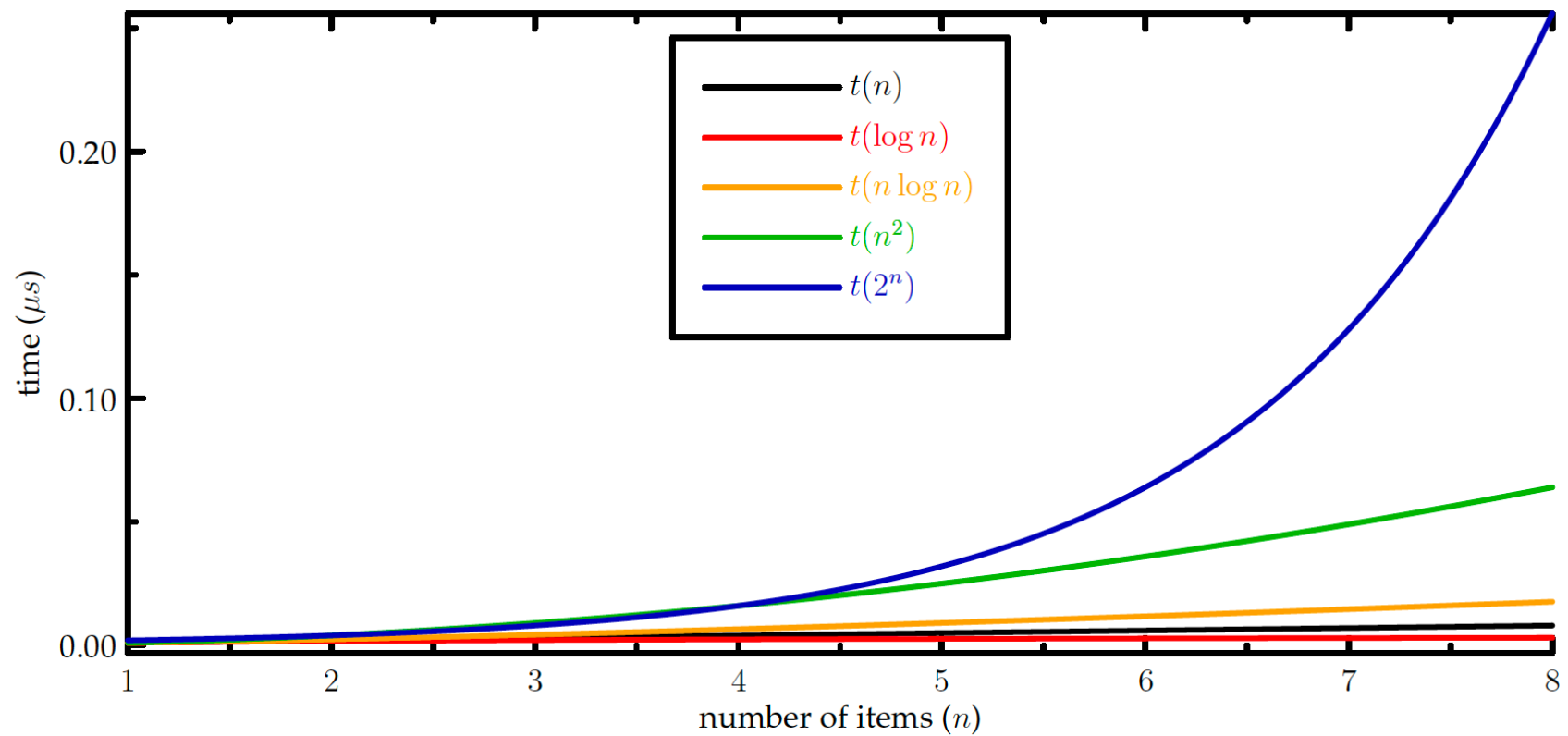


Figure 1: Notice that the 2^n is already taking way more time at $n = 8$. It only gets worse.

n up to 20

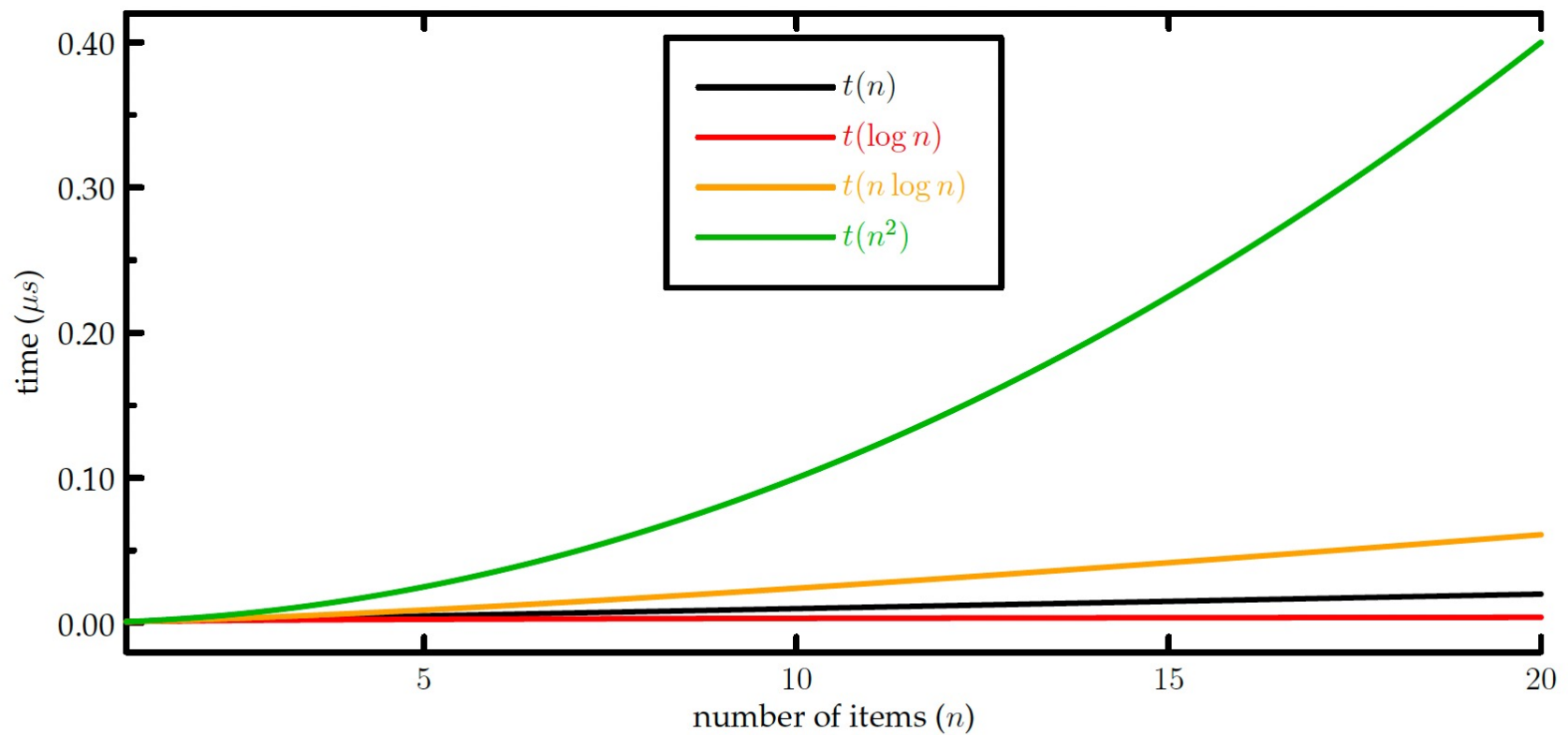
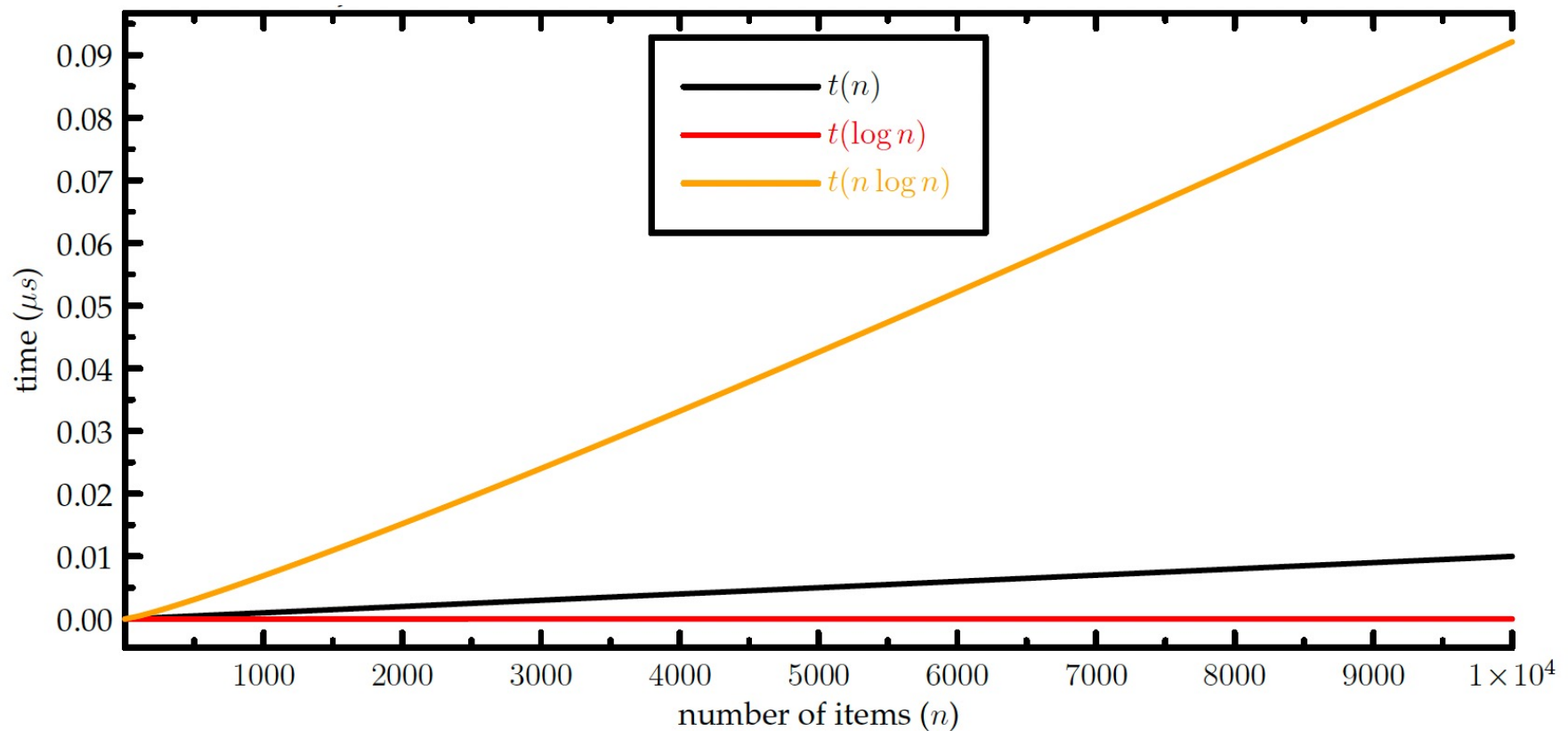


Figure 2: 2^n has been hidden because it dominates all the others. Notice that n^2 is already starting to do the same.

N up to 10000:

nlogn is taking way more time than logn now





What is “size of input”

- Searching an array:
 - The size of input is the number of elements in the array
- Merging two arrays:
 - The size of input is the total number of elements in the two arrays.
- Computing nth factorial,
 - The size is the given n.
- The size is the parameter that has an effect on the time (or space) required. There can be multiple such parameters and we may want to look at each of them separately.



Demo of Max Sub Sum problems.



Exact values

- It is sometimes possible to compute exact time and space requirements (e.g., in assembly language): how many bytes, how many cycles
- However, often the exact sequence is not known in advance. It may depend on some data structure provided at runtime.
- For high-level language, we have far less knowledge about exactly how long each operation will take.
 - Compiler will do optimization. E.g. We do not know what exactly the compiler does for $x < 10$, or $x \leq 9$.
- Our complexity analysis will use *MAJOR* simplifications.




Input size and running time

- Input size is indicated by a number n
- Running time is a function of n .
- *For example:*
constant, n , $n \log n$, n^2 , n^3
- We do simplification, instead of having $f(n)$ such as “ $18 + 3n(\log n^2) + 5n^3$ ”



Constant time

- There is some constant c such as this operation always takes c units of time.
- A statement takes constant time if
 - It does not include a loop
 - It does not call a function whose time is unknown or is not a constant



Constant time is (usually) better than linear time

■ Two algorithms:

- Algorithm A takes 5000 time units.
- Algorithm B takes $100n$ time units.

■ Which is better?

- B is better if our problem size is small: $n < 50$.
- A is better for larger problems, with $n > 50$.
- We usually care about very large problems since small problems are quick anyway.



Simplify the formula

- If $T(n) = kn + c$,
 - Constant c gets less important when n get larges.
 - K is usually a result of code optimization, not a better algorithm
 - Bottom line, we can ignore the constants k and c , and only care about n !
- In general, we can forget about the lower order functions, and still get the useful analysis results.
 - Simplify “ $18 + 3n(\log n^2) + 5n^3$ ” $\rightarrow n^3$

Order of Magnitude Notation – Big O notation

- Asymptotic Complexity - how running time scales as function of size of input
 - We usually only care about order of magnitude of scaling -- We can forget about the lower order functions.
- BIG-O
 - $T(n) = O(f(n))$
 - $f(n)$ is an Upper bound of $T(n)$
 - Exist constants c and n_0 such that
$$T(n) \leq c f(n) \quad \text{for all } n \geq n_0$$

What it means:

When n is big enough, we can just use the upper bound $f(n)$ to indicate the time complexity $T(n)$. We use Big O to notate this simplification.

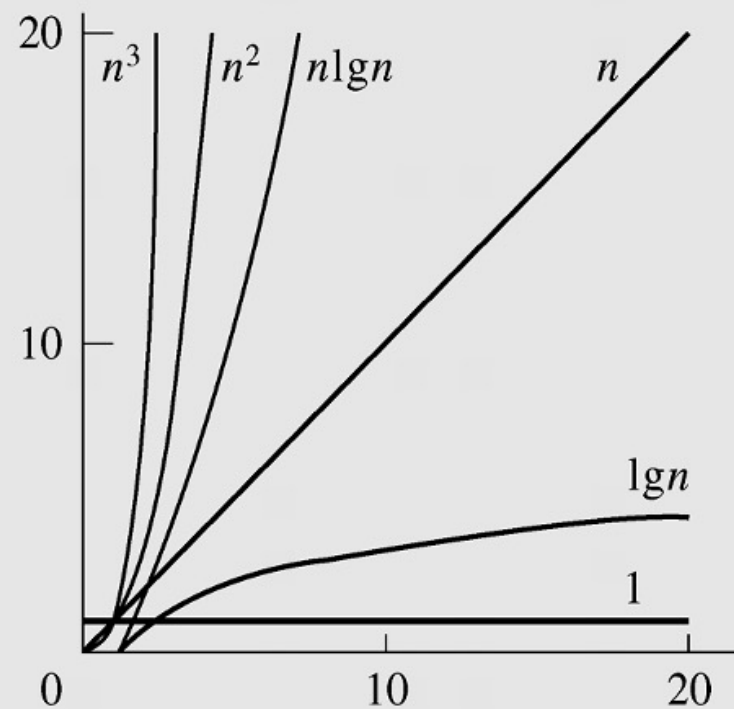
Common Cost Functions

Slowest Growth, best

constant:	$O(c)$ or $O(1)$
logarithmic:	$O(\log n)$
linear:	$O(n)$
log-linear:	$O(n \log n)$
quadratic:	$O(n^2)$
cubic:	$O(n^3)$
exponential:	$O(c^n)$ (c is a constant)

Fastest Growth, worst

Growth pattern of common functions





Example of Big O

■ $n^2 + 100n = O(n^2)$ -- WHY?

$$(n^2 + 100n) \leq 2n^2 \quad \text{for } n \geq 100$$

$$\rightarrow n_0 = 100, c = 2$$

$T(n) = O(f(n))$: Exist constants c and n_0 such that

$$T(n) \leq c f(n) \quad \text{for all } n \geq n_0$$

$T(n)$ grows no faster than $f(n)$.



Rules that simplify the analysis

■ Eliminate low order terms

$$4n + 5 \Rightarrow 4n$$

$$0.5 n \log n - 2n + 7 \Rightarrow 0.5 n \log n$$

$$2^n + n^3 + 3n \Rightarrow 2^n$$

■ Eliminate constant coefficients

$$4n \Rightarrow n$$

$$0.5 n \log n \Rightarrow n \log n$$

$$\log n^2 = 2 \log n \Rightarrow \log n$$

$$\log_3 n = (\log_3 2) \log n \Rightarrow \log n$$



Some facts

- One way equality:
 - $0.5n^2 + n = O(n^2)$, but not $O(n^2) = 0.5n^2 + n$.
 - (otherwise we may get $1/4n^2 = 1/2n^2 + n$.)
- If $f(n)$ is $O(h(n))$, and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$



Analyzing Code

- C++ operations
 - constant time
- consecutive stmts
 - sum of times
- conditionals
 - sum of branches, condition
- loops
 - sum of iterations
- function calls
 - cost of function body
- recursive functions
 - solve recursive equation


A loop that calculates the sum

```
int sum(int n)
{
    int res;
    res = 0; --- 1 unit for assignment
    for(int i=1; i<=n; i++) ← 1 assignment + (n+1) comparison + n increment
        res += i;      ← 2n
    return res; -- 1
}
```

■ $T(n) = 1 + (1 + n + 1 + n) + 2n + 1 = 4n + 4$ so,

$$T(n) = 4n + 4 = O(n).$$

■ Bottomline: The loop is what matters. Others are obviously of a lower order.



Loop

- The running time is at most the time of the statements inside the loop multiply the number of iterations

```
for  $i = 1$  to  $n$  do
```

```
     $sum = sum + 1$  (assume unit  
time 1)
```

- $T(n) = n = O(n)$



Nested Loops

```
for i = 1 to n do  
    for j = 1 to n do  
        sum = sum + 1
```

$$\sum_{i=1}^n \sum_{j=1}^n 1 = \sum_{i=1}^n n = n^2$$

$$T(n) = O(n^2)$$



Nested Dependent Loops

```
for i = 1 to n do  
    for j = i to n do  
        sum = sum + 1
```

$$T(n) = n + (n-1) + \dots + 1 = n(n+1)/2$$

Drop the lower order terms: $T(n) = 0.5 n^2 + 0.5n = O(n^2)$



Conditionals

- Conditional

`if C then S_1 else S_2`

$$\text{time} \leq \text{time}(C) + \text{Max}(\text{time}(S_1), \text{time}(S_2))$$



Consecutive Statements

```
for  $i = 1$  to  $n$  do  
     $sum = sum + 1$   
for  $i = 1$  to  $n$  do  
    for  $j = 1$  to  $n$  do  
         $sum = sum + 1$ 
```

The complex one counts.

$$T(n) = n + n^2 = O(n^2)$$



Summary: Analyzing Code

- C++ operations - constant time
- consecutive statements - sum of times
- conditionals - sum of branches, condition
- loops - sum of iterations
- function calls - cost of function body
- recursive functions (advanced) - solve recursive equation




Go back to Max Subsequence Sum

- Understand the codes and their complexity, especially the cubic, quadratic and linear.



Kinds of Analysis

- Running time may depend on **actual data input**, not just **length of input**
- Distinguish
 - **worst case** (useful, sometimes this is even what we want, e.g. for time-critical operations)
 - **best case** (often not very useful)
 - **average case** (hard to define and maybe difficult to find)
 - assumes some probabilistic distribution of inputs



Examples for best, worse, avg – linear search (unordered sequence)

```
void lfind(int x, int a[], int n)
{
    for (i=0; i<n; i++)
        if (a[i] == x)
            return;
}
```

- Best Case?
- Worst case?
- Average case?



Answer:

- Best case: 1
- Worst case: n
- Average case:
 - Considers equal probability for each case
 - $1/n (1+2+\dots+n) = (n+1)/2.$



End

- Useful website;
- <https://www.bigocheatsheet.com>