General Rules

- Rule 1 - for loops:

  The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.

- Rule 2 – nested loops:

  Analyze these inside out. The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all loops.

- Example:

```
for( i = 0; i < n; i++)
 for( j = 0; j < n;  j++)
     k++;
```

=>$O(n^2)$

General Rules

- Rule 3 – Consecutive Statements:

  These just add (the maximum is the one that counts).

  Example:

```
for( i = 0; i < n; i++)                    => O(n)
        a[ i ] = 0;
for( i = 0; i < n; i++)                    =>O(n^2)
    for( j = 0; j < n;  j++)
        a[ i ] += a[ j ] + i + j;  Total =>O(n^2)
```

- Rule 4 – if / else:

  For the fragment

```
if( condition )
  S1
else
  S2
```

  the running time of an if/else statement is never more than the running time of the test plus the larger of the running time of S1 and S2.

Analyzing Recursive Functions

- Example 1:

```
long  factorial( int n )
{
      if( n <= 1 )
       return 1;
      else
       return n * factorial( n-1 );
    }
```

=>O(n)

Poor use of recursion => can be easily transformed into a simple loop

Analyzing Recursive Functions

- Example 2:

```
long  fib( int n )
{
     if( n <= 1 )
          return 1;
     else
          return fib( n-1 ) + fib( n-2 );
 }
```

$T(n) = T(n-1) + T(n-2) + 2, \quad T(0) = T(1) = 1$

(2 => test and addition)

Easy to show that $T(n) >= fib(n)$

Easy to show that for $n > 4$, $fib(n) >= (3/2)^n$

=> $T(n) >= fib(n) >= (3/2)^n$

=> $T(N) = \Omega((3/2)^n)$ => exponential

Reason: a lot of redundant work!

Example: Maximum Subsequence Sum Problem (MSSP)

- MSSP:

  Given (possibly negative) integers $A_1, A_2, \ldots, A_N$, find the maximum value of:

  $$\sum_{k=i}^{j} A_k$$

  For convenience, the maximum subsequence sum is 0 if all the integers are negative.

- Example:

  Input: -2, 11, -4, 13, -5, -2

  Output: 20 (A_2 through A_4)

MSSP: Algorithm 1

```
        int maxSubSum1( const vector<int> & a )
        {
/* 1*/          int maxSum = 0;

/* 2*/          for( int i = 0; i < a.size( ); i++ )                    =>N
/* 3*/              for( int j = i; j < a.size( ); j++ )                =>N-i
                    {
/* 4*/                  int thisSum = 0;

/* 5*/                  for( int k = i; k <= j; k++ )                   =>j-i+1
/* 6*/                      thisSum += a[ k ];

/* 7*/                  if( thisSum > maxSum )
/* 8*/                      maxSum = thisSum;

                    }
/* 9*/              return maxSum;
        }                                                   Total => O(N^3)
```

MSSP: Algorithm 1

- How many times line 6 is executed?

$$S=\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}\sum_{k=i}^{j}1$$

$$S=\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}(j-i+1)=\sum_{i=0}^{N-1}\frac{(N-i+1)(N-i)}{2}=\sum_{i=1}^{N}\frac{(N-i+1)(N-i+2)}{2}$$

$$¿\frac{1}{2}\sum_{i=1}^{N}i^2-\left(N+\frac{3}{2}\right)\sum_{i=1}^{N}i+\frac{1}{2}(N^2+3N+2)\sum_{i=1}^{N}1$$

$$¿\frac{1}{2}\frac{N(N+1)(2N+1)}{6}-\left(N+\frac{3}{2}\right)\frac{N(N+1)}{2}+\frac{1}{2}(N^2+3N+2)N$$

$$=\frac{N^3+3N^2+2N}{6}$$

MSSP: Algorithm 1

- How many times line 6 is executed?

$$S = \frac{N^3 + 3N^2 + 2N}{6}$$

=> $\Theta(N^3)$

A lot of unnecessary computations are done in the algorithm!

Idea: use

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

MSSP: Algorithm 2

```
        int maxSubSum2( const vector<int> & a )
        {
/* 1*/        int maxSum = 0;

/* 2*/        for( int i = 0; i < a.size( ); i++ )
              {
/* 3*/            int thisSum = 0;
/* 4*/            for( int j = i; j < a.size( ); j++ )
                  {
/* 5*/                    thisSum += a[ j ];
/* 6*/                    if( thisSum > maxSum )
/* 7*/                            maxSum = thisSum;
                  }
              }
 /* 9*/       return maxSum;
        }
```

Total => $O(N^2)$

MSSP: Algorithm 3

- Divide-and-conquer strategy:
  - Divide: Split the problem into two roughly equal subproblems, which are solved recursively.
  - Conquer: patch together the two solutions to obtain the solution of the whole problem.
- The maximum subsequence sum can be in one of the following places:
  - Left half of the input: easily solved recursively
  - Right half of the input: easily solved recursively
  - In both halves: find the largest sum in the first half that includes the last element in the first half, and the largest sum in the second half that includes the first element in the second half. Add these two sums.
  - Example:

    4 -3 5 -2 || -1 2 6 -2 => mssp = 4 + 7 = 11

MSSP: Algorithm 3

```
        int maxSumRec( const vector<int> & a, int left, int right )
        {
/* 1*/  if( left == right ) // Base case
/* 2*/  if( a[ left ] > 0 )
/* 3*/  return a[ left ];
        else
/* 4*/  return 0;

/* 5*/  int center = ( left + right ) / 2;
/* 6*/  int maxLeftSum = maxSumRec( a, left, center );                  //recursive call
/* 7*/  int maxRightSum = maxSumRec( a, center + 1, right );   //recursive call

/* 8*/  int maxLeftBorderSum = 0, leftBorderSum = 0;       //check centered sequences
/* 9*/  for( int i = center; i >= left; i-- )
        {
/*10*/  leftBorderSum += a[ i ];
/*11*/  if( leftBorderSum > maxLeftBorderSum )
/*12*/  maxLeftBorderSum = leftBorderSum;
        }

/*13*/  int maxRightBorderSum = 0, rightBorderSum = 0;
/*14*/  for( int j = center + 1; j <= right; j++ )
        {
/*15*/  rightBorderSum += a[ j ];
/*16*/  if( rightBorderSum > maxRightBorderSum )
/*17*/  maxRightBorderSum = rightBorderSum;
        }

/*18*/  return max3( maxLeftSum, maxRightSum,
/*19*/  maxLeftBorderSum + maxRightBorderSum );
        }
```

MSSP: Algorithm 3 Analysis

- T(N) = time to solve MSSP of size N.
- If N =1 => constant time T(1) =1
- Lines 8 to 18 take O(N)
- Lines 6 and 7 solve two MSSP of size N/2 each => T(N/2)
- Total time:

  T(1) = 1

  T(N) = 2T(N/2) + O(N)
- T(N) = O(N log N)
- This analysis is valid for N power of 2.
- If N is not a power of 2 => more complex analysis but the Big-Oh remains unchanged.

MSSP: Algorithm 3 Analysis

Master Theorem:

$T(N) = a\,T(N/b) + O(N^d)$

for some constants a >0, b>1 and d >= 0 then,

1) $T(N) = O(N^d)$ if $d > \log_b a$

2) $T(N) = O(N^d \log N)$ if $d = \log_b a$

3) $T(N) = O(N^{\log_b a})$ if $d < \log_b a$

Algorithm 3 analysis:

- Total time:

$T(N) = 2T(N/2) + O(N)$

a = 2, b =2, d = 1

$\log_b a = \log_2 2 = 1 = d$ ==> case 2

$T(N) = O(N \log N)$

MSSP: Algorithm 4

- Based on the following observation:
  - If a[i] is negative then it cannot possibly be the start of the optimal subsequence because any subsequence that starts with it can be improved by beginning with a[i+1].
  - Any negative subsequence cannot possibly be a prefix of the optimal subsequence => advance i to j+1 if the sum from a[i] to a[j] is negative.
- The algorithm makes only one pass through the data.
- It is an on-line algorithm i.e. at any point in time can give a correct answer to the problem for the data it has already read.

MSSP: Algorithm 4

```
        int maxSubSum4( const vector<int> & a )
        {
/* 1*/          int maxSum = 0, thisSum = 0;

/* 2*/          for( int j = 0; j < a.size( ); j++ )
                {
/* 3*/                  thisSum += a[ j ];

/* 4*/                  if( thisSum > maxSum )
/* 5*/                          maxSum = thisSum;
/* 6*/                  else if( thisSum < 0 )
/* 7*/                          thisSum = 0;
                }

/* 8*/          return maxSum;
        }
```

=> Total time = O(N)

MSSP: History

- 1977: Ulf Grenander (Brown Univ.)

Pattern recognition problem:

find the maximum sum over all rectangular regions of a given 2D array of real numbers (maximum likelihood estimator)

- Example:

-1	2	-3	5	-4	-8	3	-3
2	-4	-6	-8	2	-5	4	1
3	-2	9	-9	-1	10	-5	2
1	-3	5	-7	8	-2	2	-6

- Grenander devised an $O(N^6)$ algorithm similar to Algorithm 1 for 1D MSSP.
- Grenander reduced the problem to 1D to gain more insights on how to develop efficient algorithms.
- Finding algorithms for 2D MSSP is sometimes asked in job interviews at Google.

MSSP: History

1D MSSP Problem:

- Grenander developed the $O(N^2)$ algorithm (Algorithm 2).
- Grenander described the problem to Michael Shamos from CMU who overnight designed the O(N logN) algorithm (Algorithm 3).
- Shamos described the problem at a CMU seminar where Jay Kadane designed the O(N) algorithm (Algorithm 4) within a minute.
- There is no faster algorithm than Algorithm 4!

2D MSSP Problem:

- $O(N^3)$ algorithm, extending Kadane's algorithm (Algorithm 4)
- Open Problem: Finding an $O(N^2)$ algorithm for 2D MSSP!

Example: Binary Search

- Binary search:

  Given an integer *X* and integers

  $A_0, A_1, \ldots, A_{N-1}$, which are presorted and already in memory, find *i* such that $A_i = X$ or return $i = -1$ if *X* is not in the input.

- Obvious solution:

  scan through the list from left to right

  => T(N) = O(N)

- Better strategy:

  If x < middle element => search left side

  If x > middle element => search right side

Example: Binary Search

```
        template <class Comparable>
        int binarySearch( const vector<Comparable> & a, const Comparable &
        x )
        {
/* 1*/              int low = 0, high = a.size( ) - 1;

/* 2*/              while( low <= high )
                    {
/* 3*/                      int mid = ( low + high ) / 2;

/* 4*/                      if( a[ mid ] < x )
/* 5*/                              low = mid + 1;
/* 6*/                      else if( a[ mid ] > x )
/* 7*/                              high = mid - 1;
                            else
/* 8*/                          return mid; // Found
                     }

/* 9*/                 return NOT_FOUND; // NOT_FOUND is defined as -1
        }
```

T(N) = T(N/2) + O(1) => T(N) = O(log N)

Example: Euclid's Algorithm

- Computes the greatest common divisor (gcd) of two integers.
- gcd = largest integer that divides both.
- Example: gcd(50,15) = 5
- The algorithm computes gcd(M,N) assuming M >= N. If N > M swaps them in the first iteration.
- It computes the remainders until 0 is reached.
- The last nonzero remainder is the answer.
- Example:

  M = 1,989, N = 1,590

  the sequence of remainders: 399, 393, 6, 3, 0

  => gcd(1989, 1590) = 3.
- Theorem: After two iterations the remainder is at most half of its original value.
- The number of iterations is at most 2 log N = O(log N)
- If N is an n-bit integer => O(n)
- Each iteration involves an integer division (which takes $O(n^2)$)

  => Total time = $O(n^3)$

Example: Euclid's Algorithm

```
long gcd( long m, long n )
{
    while( n != 0 )
       {
         long rem = m % n;
            m = n;
         n = rem;
     }
       return m;
}
```

Example: Exponentiation

- Compute X^N where X, N are integers
- Obvious algorithm: use N-1 multiplications => O(N)
- Better algorithm:
  - If N is even => $X^N = X^{N/2} X^{N/2}$
  - If N is odd => $X^N = X^{(N-1)/2} X^{(N-1)/2} X$
- At most two multiplications are required to halve the problem (N odd).
- The number of multiplications is 2 log N => T(N) = O(log N)

Example: Exponentiation

```
long pow( long x, int n )
{
    if( n == 0 )
        return 1;
    if( n == 1 )
        return x;
    if( isEven( n ) )
        return pow( x * x, n / 2 );
      else
        return pow( x * x, n / 2 ) * x;
}
```