# Maximum Contiguous Subsequence Sum

- Given $A_1, A_2, A_3, \dots, A_N$,
- Find a contiguous subsequence with maximum sum.

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

- If MCSS < 0, then MCSS=0

{-2,**11,-4,13**,-5,2}  MCSS=20

{1,-3,**4,-2,-1,6**}  MCSS=7



## Exhaustive Search $O(N^3)$

- Find a contiguous subsequence with maximum sum.
  1. for each i ➔ $N$
  2. for each j ➔ $N$
  3. compute $\sum_{k=i}^{j} A_k$ and support Max among the sums ➔ $N$
- Simple ➔ easy to program, less probability of programming error
- Not efficient O( $N^3$ )

*Theorem 1 :* The number of ordered triples (*i, j, k*) that satisfy $1 \le i \le j \le k \le N$ is *N(N+1)(N+2)/6* .

*Proof :*



```
// Cubic maximum contiguous subsequence sum algorithm.
// Comparable: must have constructor from int, must have
//       += and > defined, must have copy constructor
//       and operator=, and must have all properties
//       needed for vector.
// seqStart and seqEnd represent the actual best sequence.

template <class Comparable>
Comparable maxSubsequenceSum1( const vector<Comparable>  & a, int & seqStart, int & seqEnd )
{
        int n = a.size( );
        Comparable maxSum = 0;
        for( int i = 0; i < n; i++ )
                for( int j = i; j < n; j++ )
                {
                        Comparable thisSum = 0;
                        for( int k = i; k <= j; k++ )
                                thisSum += a[ k ];
                        if( thisSum > maxSum )
                        {
                                maxSum = thisSum;
                                seqStart = i;
                                seqEnd = j;
                        }
                }
        return maxSum;
}
```



## How to improve it to $O(N^2)$

- Note that each time we compute a sum, we throw away the previous sum
- Can we save it ➔ Yes ➔ How?

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

- Having $\sum_{k=i}^{j-1} A_k$ requires only one more step to get $\sum_{k=i}^{j} A_k$



```
// Quadratic maximum contiguous subsequence sum algorithm.
// Comparable: must have constructor from int, must have
//       += and > defined, must have copy constructor
//       and operator=, and must have all properties
//       needed for vector.
// seqStart and seqEnd represent the actual best sequence.

template <class Comparable>
Comparable maxSubsequenceSum2( const vector <Comparable> & a, int & seqStart, int & seqEnd )
{
        int n = a.size( );
        Comparable maxSum = 0;
        for( int i = 0; i < n; i++ )
        {
                Comparable thisSum = 0;
                for( int j = i; j < n; j++ )
                {
                        thisSum += a[ j ];
                        if( thisSum > maxSum )
                        {
                                maxSum = thisSum;
                                seqStart = i;
                                seqEnd = j;
                        }
                }
        }
        return maxSum;
}
```



## How to improve it to $O(N)$

- We need to remove another loop, how?
- Can we eliminate those subsequences that we know they don't satisfy the conditions?
- To do that let,
  - $A_{i,j}$ be a subsequence $A_i, A_{i+1}, \dots, A_j$
  - $S_{i,j}$ be $\sum A_i, A_{i+1}, \dots, A_j$
- Can we identify those subsequences that could not be maximum?

*Theorem 2:* Let $A_{i,j}$ be any sequence with $S_{i,j} < 0$.
If $q > j$, then $A_{i,q}$ is not the maximum contiguous subsequence.

*Proof:*

*Theorem 2:* Let $A_{i,j}$ be any sequence with $S_{i,j} < 0$
If $q > j$, then $A_{i,q}$ is not the maximum contiguous subsequence.

*Proof:*

- *i, i+1,..., j, j+1,..., q*
- $S_{i,q} = S_{i,j} + S_{j+1,q}$
- Since $S_{i,j} < 0 \Rightarrow S_{i,q} < S_{j+1,q} \Rightarrow$

$A_{i,q}$ is not a maximum contiguous subsequence

| $i$ | $j$ $j+1$ | $q$ |
|---|---|---|
| <0 | $S_{j+1,q}$ | |
| $< S_{j+1,q}$ | | |

- If we add an additional test, *if thisSum* < 0, then we can break the inner loop.
  - If a subsequence sum is negative, it cannot be part (prefix) of MCSS.



*Theorem 3:* For any $i$, let $A_{i,j}$ be the first sequence with $S_{i,j} < 0$,
then, for any $i \le p \le j$ and $p \le q$, $A_{p,q}$ either is not a maximum contiguous subsequence or is equal to an already seen maximum contiguous subsequence.

*Proof:*

- If $p = i \Rightarrow i \le j$ and $i < q$, Theorem 2 applies. Why?
  ➔ part of the sum is negative.
- Else $(i < p)$, as in Theorem 2, $S_{i,q} = S_{i,p-1} + S_{p,q}$.

Since *j* is the lowest index such that $S_{i,j} < 0 \Rightarrow$
$S_{i,p-1} \ge 0 \Rightarrow S_{p,q} \le S_{i,q}$.

If $q > j$, then Theorem 2 implies that $A_{i,q}$ is not MCSS, so neither is $A_{p,q}$

| $i$ | $j$ $j+1$ | $q$ |
|---|---|---|
| $S_{i,q}$ | | |
| $\ge 0$ | $\le S_{i,q}$ | |
| $p-1$ | $p$ | |



If $q \le j$, then $S_{p,q} \le S_{i,q}$ implies that the subsequence $A_{p,q}$ has a sum equal to, at most, that of the already seen subsequence $A_{i,q}$.

| $i$ | $q$ | $j$ |
|---|---|---|
| $S_{i,q}$ | | |
| $\ge 0$ | $\le S_{i,q}$ | |
| $p-1$ | $p$ | |

- Conclusions: If we detect a negative sum, we can move *i* all the way past *j*.



```
// Linear maximum contiguous subsequence sum algorithm.
// Comparable: must have constructor from int, must have
//              += and > defined, must have copy constructor
//             and operator=, and must have all properties
//             needed for vector.
// seqStart and seqEnd represent the actual best sequence.

template <class Comparable>
Comparable maxSubsequenceSum3( const vector<Comparable>  & a, int & seqStart, int & seqEnd )
{
        int n = a.size( );
        Comparable thisSum = 0;
        Comparable maxSum = 0;
        for( int i = 0, j = 0; j < n; j++ )
        {
                thisSum += a[ j ];
                if ( thisSum > maxSum )
                {
                        maxSum = thisSum;
                        seqStart = i;
                        seqEnd = j;
                }
                else if ( thisSum < 0 )
                {
                         i = j + 1;
                        thisSum = 0;
                }
        }
        return maxSum;
}
```

- What is the time complexity? How many loops? ➔ *O(N)*



# General Big-Oh Rules

- Algorithm analysis used to study resource trade-off analysis is generally a theoretical issue.

***Def.1***. (Big-Oh) The function $T(N) = O(F(N))$ if there is a positive constant *c* and $N_0$ such that $T(N) \le cF(N)$ when $N \ge N_0$. ➔ Growth of $T(N) \le$ growth of $F(N)$.

***Def.2***. (Big-Omega) The function $T(N) = \Omega(F(N))$ if there is a positive constant *c* and $N_0$ such that $T(N) \ge cF(N)$ when $N \ge N_0$. ➔ Growth of $T(N) \ge$ growth of $F(N)$.

***Def.3***. (Big-Theta) The function $T(N) = \Theta(F(N))$ if and only if $T(N) = O(F(N))$ and $T(N) = \Omega(F(N))$. ➔ Growth of $T(N) =$ growth of $F(N)$.

***Def.4***. (Little-Oh) The function $T(N) = o(F(N))$ if and only if $T(N) = O(F(N))$ and $T(N) \ne \Theta(F(N))$. ➔ Growth of $T(N) <$ growth of $F(N)$.

## More definitions

- Worst-case bound: Guaranteed bound over all inputs
- Average-case bound: Average bound over all inputs
- Note that $1{,}000N > N^2$ for small $N$ $(N < 1{,}000)$ but $N^2$ grows at a faster rate



## Some Important Rules

- ***Rule 1***: If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = Max(O(f(N)), O(g(N)))$
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$
- ***Rule 2***: If $T(N)$ is a polynomial of degree $k$, then $T(N) = \Theta(N^k)$
- ***Rule 3***: $\log^k N = O(N)$ for any constant $k$ ➔ logarithmic functions grow very slow

| | |
|---|---|
| $c$ | constant |
| $\log N$ | logarithmic |
| $\log^2 N$ | log-squared |
| $N$ | linear |

| | |
|---|---|
| $N \log N$ | N times log N |
| $N^2$ | quadratic |
| $N^3$ | cubic |
| $2^N$ | exponential |

It is a bad style to include constants or low order terms inside a Big-Oh (don't say $T(N) = O(2N^2)$ or $T(N) = O(N^2 + N)$ ➔ both are $O(N^2)$).

# The Logarithm

***Def***. For any $B, N > 0,\ \log_B N = K \Leftrightarrow B^K = N$

- $B$ is the base, in CS often $B = 2$

***Theorem***. For any constant $B > 1,\ \log_B N = O(\log N)$

***Proof.*** $\log_B N = K \Rightarrow B^K = N$

Let $C = \log B \Rightarrow 2^C = B \Rightarrow B^K = (2^C)^K = N \Rightarrow$
$2^{CK} = N \Rightarrow \log N = CK = C \log_B N.$

Therefore $\log_B N = (\log N)/(\log B).$

***Homework:*** Make sure to review and understand the logarithmic rules from Calculus. Here are a few useful rules, try their proof

- $\log_B XY = \log_B X + \log_B Y$
- $\log_B X \times \log_X B = 1 \Leftrightarrow \log_B X = 1/\log_X B$
- $\log_B X = \log_D X / \log_D B$ for any base $D$ (see Theorem).



# A Few Logarithmic Problems

- How many bits are required to represent N consecutive numbers?

$$\lceil \log_2 N \rceil$$

- How many times X should be doubled before it reaches N?

$$2^K X = N \Rightarrow K = \lceil \log_2 N / X \rceil$$

- How many times X should be halved before it reaches N?

$$2^{-K} X = N \Rightarrow K = \lceil \log_2 X / N \rceil$$



# Static searching

- The data is never altered, assuming stored in an array

## Sequential Search

- Unsuccessful search requires *N* tests ➔ *O(N)*
- On average search requires *N/2* tests ➔ *O(N)*

## Binary search

- Assuming data is sorted
- Test is performed from the middle of the array



```
// BinarySearch: Return position of x in sorted array a
//          or NOT_FOUND if item is not found.
// Comparable: must have operator< and operator== and all
//          properties needed for vector.

template <class Comparable>
int binarySearch( const vector<Comparable>  & a, const Comparable & x )
{
          int low = 0;
          int high = a.size( ) - 1;
          int mid;
          while( low <= high )
          {
                    mid = ( low + high ) / 2;
                    if( a [ mid ] < x )
                              low = mid + 1;
                    else if( a [ mid ] > x )
                                   high = mid-1;
                         else
                                   return mid;
          }
          return NOT_FOUND;
}
```



- Analysis
  - Unsuccessful search requires $\lfloor \log N \rfloor + 1$ , Why?

    The final half consists of 0 elements
  - Worst case successful search requires $\lfloor \log N \rfloor$ , Why?

    The final half consists of 1 element
  - Average case is one iteration better, Why?

    1/ 2 of elements require the worst case for their search

    1/ 4 of elements save one iteration

    1/ 8 of elements save two iterations
  - Computing weighted average results in $\lfloor \log N \rfloor - 1$ which is *O(logN)*



# Interpolation Search

- While Binary search is fast, *Interpolation Search* sometimes gives better speed
- Note that if data is on disk, it has to be brought to memory
- The conditions that make Interpolation Search acceptable
  - data access is time consuming
  - while sorted, the data is uniformly distributed
- Instead of calculating the ***mid*** in Binary Search, we compute ***next***, the next item we will try to access

$$next = low + \left\lceil \frac{x - a[low]}{a[high] - a[low]} \times (high - low - 1) \right\rceil$$

- Note that this calculation of ***next*** is more costly than ***mid*** in Binary Search in terms of multiplication and division

## Checking An Algorithm Analysis and Limitations

- *Empirically* observed running time versus *analytical* predictions
- Trade-offs
  - # of instructions
  - # of operations
  - data distribution
  - access cost
  - sorting cost
- Note that elimination of the lower orders may have significant impact, particularly when $N$ is small
- The same for constant $c$ and the coefficient



## Homework

1. Problem 6.3, page 223 of the text
2. Problem 6.4, page 223 of the text
3. Problem 6.11, page 223 of the text
4. Problem 6.24, page 223 of the text