Once you know your algorithm works, you then need to make sure it doesn't take too long or use too much space.

Lecture 2
Algorithmic Analysis

Algorithmic analysis is the computation of how many resources are needed by the algorithm.

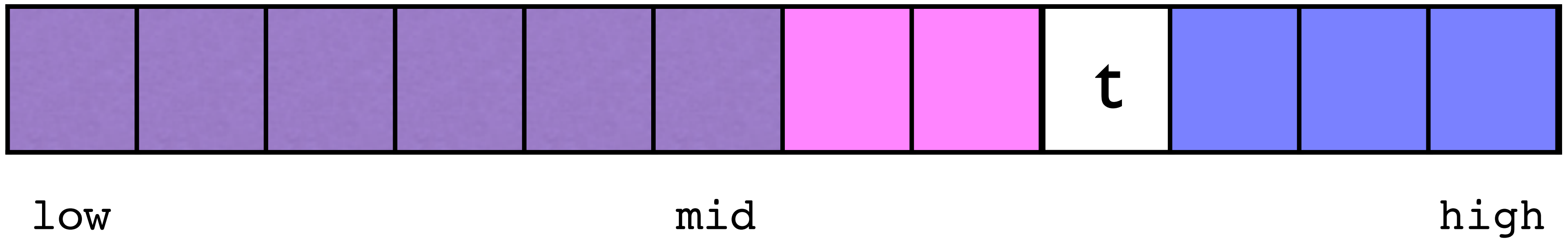
Comments

- For the same problem (e.g. sorting, searching) there are different algorithms, some of which are substantially better than others
- Selection of right data structure and/or preprocessing may allow substantial improvement in running time (but may have a one-time preprocessing cost).

Static Searching (data doesn't change)

If key x is in array a, return its position; else indicate that it's not there.

- Recall Sequential search: O(n)
- If array is *sorted*, we can search much more efficiently by using Binary Search (our version is using 2-way comparisons)
- Analysis: Repeated Halving : O(log n)



Binary Search for x = s

```
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
            high = mid;
    }
    return ( low == high && a[ low ] == x ) ? low : NOT_FOUND;//NOT_FOUND = -1
}
```

Worst Case Running Time?

After the first comparison how many items remain? $n/2$

After the second comparison how many items remain? $(n/2)/2 = n/2^2$

After the third comparison how many items remain? $(n/2^2)/2 = n/2^3$

...

After the kth comparison how many items remain? $n/2^k$

How many comparison till one item remains? $\lceil \log(n) \rceil$

Total number of comparisons? $\lceil \log(n) \rceil + 1$

Using Big-Oh notation: $O(\log(n))$

$\log(n) = \log_2(n)$ in this class

Worst Case

# names	sequential search	Binary SearchB
512	512	9+1
1024	1024	10+1
2048	2048	11+1
4196	4196	12+1
8192	8192	13+1
16384	16384	14+1
32768	32768	15+1
65536	65536	16+1
...	...	...
16,777,216	16,777,216	24+1
...	...	...
268,435,456	268,435,456	28+1

The logarithm functions appears frequently in the analysis of algorithms and data structures

Computers store integers in binary. In computer science the most common base is 2. In computer science if no base is given it is assumed to be 2.

$\log n = \log_2 n$

The Logarithm

$b^x = n$ if and only if $x = \log_b n$

b is the base of $\log_b n$

$\log_b 1 = 0$

We can approximate the logarithm function, $\log_b n$, by the number of times we can divide n by b till we get less than or equal to one

$\log_2 8 = 3$	$8/2/2/2 = 1$	$2^3 = 8$
$\log_3 81 = 4$	$81/3/3/3/3 = 1$	$3^4 = 81$
$\log_3 79 \sim 4$	$79/3/3/3/3 < 1$	$3^4 > 79$

This approximation is less than one away from the true value

Useful Facts about Logs

- $\log_b(a^c) = c \log_b a$
- $\log_b(a) = \log_c a / \log_c b$
- $\log_b n$ is $O(\log n)$

Notice, to change the base of the log you multiply by a constant!

- Log n grows very slowly:

n	log n
1	0
2	1
4	2
8	3
16	4
...	
1024	10
2^{20} (about 1 million)	20

The Base of the logarithm doesn't matter!

Big-Oh

$\log_4(n)$ is $O(\log(n))$

$\log_4(n^4)$ is $O(\log(n))$

$(\log_2(n))^3$ is $O(\log^3(n))$

Why?

Definition: (Big-Oh) T(n) is O(F(n)) if there are positive c and n_0 such that T(n)<=cF(n) when n>= n_0.

- Exponential function grows *very fast*; logarithmic function grows *very slowly*
- So programs with exponential running time run *very slowly*; programs with logarithmic running time run *very quickly*.
- Precise analysis usually involves floors or ceilings, but when we go to Big-Oh, it doesn't really matter.

Which would you prefer? (Sorry I am not offering!)

A million dollars

or

A penny doubled every day for thirty days.

Day 1: 1 penny

Day 2: 2 pennies

Day 3: 4 pennies

Day 4: 8 pennies

Day 5: 16 pennies

... till day 30

2^{29} =\$5,368,709.12

If you were given 32 homework assignments, and on the first day you did 1/2 of the problems, the next day and every day afterwards you did 1/2 of the problems, how many days until you had 1 problem left to do?

How about 512?

How about 1024?

Repeated Halving and Doubling

- `for (i = n; i > 1; i = i/2)` // log n iterations

- `for (i =1; i < n; i = i*2)` // log n iterations

Algorithms based on repeated halving (or doubling) are generally very efficient

Estimating Running Times

```
for (i = 1; i < n; i = 2*i)
    sum += a[i]
```

n	steps	O(log(n))
64	6	
128	7	
256	8	
512	9	

```
for (i = 1; i < n; i = 4*i)
    sum += a[i]
```

n	steps	O(log(n))
64	3	
128	4	
256	4	
512	5	

```
for (i = n; i > 1; i /= 4)
    sum += a[i]
```

n	steps	O(log(n))
64	3	
128	4	
256	4	
512	5	

Bits in a Binary Number!

16-bit short integer represents the integers from
-32,768 to 32,767

65,636 integers (i.e. $2^{16} = 65,636$)

If you needed to store an integer in the range from 0 to 232222 how many bits would you need?

$\log(232223) = 17.8$ bits doesn't make sense!

$\lceil 17.8 \rceil = 18$ bits

Floors and Ceilings:

$$\lceil \log(2147483654) \rceil = \lceil 31.000000000403084 \rceil = 32$$

$$\lfloor 4.9 \rfloor = 4$$

$$\lfloor 4.01 \rfloor = 4$$

$$\lceil 4.9 \rceil = 5$$

$$\lceil 4.01 \rceil = 5$$

$$\lfloor 4 \rfloor = 4$$

$$\lceil 4 \rceil = 4$$

Estimating Run times!

If an item isn't found, how many operations are performed by the code snippet?

<u>Uniform cost model</u> - assume every simple statement has a constant cost.

Linear Search

```
int index = -1;
for( i = 0; i < n; i++ )
   if( item == a[i] )
      {
         index = i;
         break;
      }
```

	n=50	n=100	n=200	n=400
	1	1	1	1
	1, 51, 50	1, 101, 100	1, 201, 200	1, 401, 400
	50	100	200	400
Total	153	303	603	1203
	153/50	303/100	603/200	1203/400

as n gets large this ratio approaches 3

$T(n) = 2 + (n + 1) + n + n = 3n + 3$

$T(2n) = 2 + (2n + 1) + 2n + 2n = 3(2n) + 3$

For a linear time algorithm when we double the input size, the number of steps the algorithm takes is roughly twice as many c_1n+c_2 ☛ $c_1(2n) + c_2 \sim c_1(2n)$

For a linear time algorithm for and input size of n', the number of steps the algorithm takes is roughly $r = n'/n$ times as many c_1n+c_2 ☛ $c_1(n') + c_2 \sim c_1(rn)$

$T(n)/n = (c_1n+c_2)/n \sim c_1$

Counting the number of steps

Uniform Cost Model
Only counting operations (not actual machine code, or the difference in time to compute the different instructions)

	n=50	n=100	n=200
`for (i=0;i<n;i++)`	1 + 51 + 50	1 + 101 + 100	1 + 201 + 200
`  for(j=0; j<n; j++)`	50(1 + 51 + 50)	100(1 + 101 + 100)	200(1 + 201 + 200)
`    sum += a[i]*a[j]`	50 * 50	100 * 100	200 * 200
Total	7702	30402	120802
	7702/2500	30402/10000	120802/40000
	= 3.0808	= 3.0402	= 3.02005

The number of steps for this code snippet:

$$T(n) = 1 + (n + 1) + n + n(1 + (n + 1) + n) + n * n = 2 + 4n + 3n^2$$

doubling the size of n: $T(2n) = 2 + 4(2n) + 3(2n)^2 = 2 + 8n + 12n^2 \sim 4T(n)$

for an input size n' **where** $r = n'/n$:

$$T(n') = 2 + 4(n') + 3(n')^2 = 2 + 4rn + 3r^2n^2 \sim r^2T(n)$$

The number of steps for an algorithm which take O(n²) time:

$$T(n) = c_1n^2 + c_2n + c_3$$

The largest term dominates as n gets large

The number of steps on an input size n' **where** $r = n'/n$:

$$T(n') = c_3 + c_2(n') + c_1(n')^2 = c_3 + c_2rn + c_1r^2n^2 \sim r^2T(n)$$

$$T(n)/n^2 = (c_1n^2 + c_2n + c_3)/n^2 \sim c_1$$

Estimating Run Times

<u>T(n), the running time of the program</u>

- Can represent the number of steps
- Can represent the time it takes a computer to execute the code
- …

If the algorithm is O(n^2),
and it takes 0.0073 seconds when $n = 2^7$
How long should it take when $n = 2^8$?

Hmm …

If $T(n) = c_1n^2 + c_2n + c_3$ is $O(n^2)$

$T(n) \sim c_1n^2$ when n is large

$T(2n) \sim c_1(2n)^2 = 2^2c_1n^2$

If $T(2^7) = 0.0073$

We approx $T(2^8) \sim 4*T(2^7) = 4*0.0073 = 0.0292$

Estimating Run Times

If the algorithm is O(n^3),
and it takes 0.0073 seconds when n = 2^7
How long should it take when n = 2^8?

Hmm ...

If $T(n) = c_1n^3 + c_2n^2 + c_3$ is $O(n^3)$
$T(n) \sim c_1n^3$ when n is large

$T(2n) \sim c_1(2n)^3 = 2^3c_1n^2$

If $T(2^7) = 0.0073$

We approx $T(2^8) \sim 8*T(2^7) = 8*0.0073 = 0.0584$

Estimating the Actual Running Time

- Program P has $O(n^k)$ running time and runs for time t_0 on input of size n_0
- How long will program take on input of size n_1?
- Compute $r = n_1/n_0$
- Program will take approximately $r^k * t_0$ on input of size n_1

Note: We get better estimates when we start with a larger number.

Estimating Run Times

For an O(n^3) time algorithm where it takes 0.0073 seconds when n = 2^7. When n = 2^8 we estimated T(2^8) is roughly 0.0584.

T(2^9) is roughly 0.4672

T(2^{10}) is roughly 3.7376

T(2^{11}) is roughly 29.9008

T(2^{12}) is roughly 239.2064

	n^3
2^7	0.0073
2^8	0.05642
2^9	0.440588
2^{10}	3.50385
2^{11}	28.0077
2^{12}	233.944

T(2^{18})=T(262144) is roughly 8589934592* 0.0073 = 62706522.5216

2.49234181522182 years

Estimating Run Times

If the algorithm is O(n^2),
and it takes 0.00024 seconds when n = 2^7
When n = 2^8 we estimated T(2^8) is roughly 0.00096.

T(2^9) is roughly 0.00384

T(2^{10}) is roughly 0.01536

T(2^{11}) is roughly 0.06144

T(2^{12}) is roughly 0.24576

T(2^{18})=T(262,144) is roughly
4194304* 0.00024 = 1006.63296

	n^2
2^7	0.00024
2^8	0.00091
2^9	0.00358
2^{10}	0.014356
2^{11}	0.058269
2^{12}	0.232687

Run times

For the linear time algorithm, $T(2^{18})=T(262,144)$ is roughly ______

	n
2^7	8E-06
2^8	1.2E-05
2^9	2.1E-05
2^{10}	3.5E-05
2^{11}	9.3E-05
2^{12}	0.000139

Checking we have the correct asymptotic function!

$T(n^3)/n^3$

$T(2^7)/(2^7)^3 = 0.0073/2097152=0.0000000034809l$

$T(2^8)/(2^8)^3 = 0.05642/16777216=0.00000000336289$

$T(2^9)/(2^9)^3 = 0.440588/134217728=0.00000000032826$

$T(2^{10})/(2^{10})^3 = 3.50385/1073741824=0.000000003263$

What if we thought it was O(n^2)?

What if we thought it was O(n^4)?

	n^3
2^7	0.0073
2^8	0.05642
2^9	0.440588
2^{10}	3.50385
2^{11}	28.0077
2^{12}	233.944

$T(n^2)/n^2$

$T(2^7)/(2^7)^2 =$ 0.00024/16384=0.0000000146484

$T(2^8)/(2^8)^2 =$ 0.00091/65536=0.0000000138855

$T(2^9)/(2^9)^2 =$ 0.00358/262144=0.0000000136566

$T(2^{10})/(2^{10})^2 =$ 0.014356/1048576=0.0000000136

	n^2
2^7	0.00024
2^8	0.00091
2^9	0.00358
2^{10}	0.014356
2^{11}	0.058269
2^{12}	0.232687

Running Times.... Which is $O(n)$, $O(n^2)$, $O(n^3)$?

n=10	0.000009	0.000004	0.000003
100	0.002580	0.000109	0.000006
1000	2.281013	0.010203	0.000031
10000	NA	1.2329	0.000317
100000	NA	135	0.003206

Max Contiguous Subsequence Problem

One problem

3 solutions!

* Ok, I really should have said we will discuss 3 of the many solutions possible to solve this problem.

Max Contiguous Subsequence Problem

- Given sequence A1,...,An of numbers find i and j such that Ai + ... + Aj is maximal.
- 1, 2, -4, 1, 2, -1, 4, -2, 1
- Max subsequence is 1,2,-1,4 whose sum is 6

When in doubt, use brute force.

-Ken Thompson, Bell Labs

Max Contiguous Subsequence Problem

{-2, 11, -4, 13}

	{}	0
	{-2}	-2
	{-2, 11}	9
	{-2, 11, -4}	5
	{-2, 11, -4, 13}	18
	{11}	11
	{11, -4}	7
Maximum	{11, -4, 13}	20
	{-4}	-4
	{-4, 13}	9
	{13}	13

Max Contiguous Subsequence Problem

Definition:

- Given sequence A1,...,An of numbers find i and j such that Ai + ... + Aj is maximal.
- A1, A2, A3, A4, A5, A6, A7, A8, A9
- 1,   2, -4, 1,   2, -1,   4, -2,   1
- Max subsequence is 1,2,-1,4 whose sum is 6

How did we know?   Computed all subsequences...

Sum(1,9)=1+2-4+1+2-1+4-2+1=4
Sum(1,8)=1+2-4+1+2-1+4-2=3
Sum(1,7)=1+2-4+1+2-1+4=5
⋮
Sum(2,9)=1+2-4+1+2-1+4-2+1=3
⋮
Sum(5,7)=2-1+4=5

Till we tried everything!

Naive Algorithm

For each pair i, j compute the sum of the elements from i to j,
Keeping track of maxsofar.

```
For each pair (i,j):
        thissum =0;
        for (k=i; k <=j; k++) thissum += a[k];
          ...
          if (thissum > maxsofar) maxsofar = thissum;
```

```
 1    2    -4    1    2    -1     4    -2    2
```

e.g., when (i,j) = (1,4), sum = 2 - 4 +1 +2 = 1

Triply nested loop: O(n^3)

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

    for( int i = 0; i < a.size( ); ++i )
        for( int j = i; j < a.size( ); ++j )
        {
         int thisSum = 0;

             for( int k = i; k <= j; ++k )
                 thisSum += a[ k ];

           if( thisSum > maxSum )
               maxSum   = thisSum;
       }

    return maxSum;
}
```

For each pair i, j
compute the sum of the elements from i to j,
keeping track of the largest one you found, maxsofar.

a	1	2	-4	2	-1	4	-2	2	-5	9	12	-3	7	11

First Algorithm

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

    for( int i = 0; i < a.size( ); ++i )
        for( int j = i; j < a.size( ); ++j )
        {
            int thisSum = 0;

            for( int k = i; k <= j; ++k )
                thisSum += a[ k ];

            if( thisSum > maxSum )
                maxSum   = thisSum;
        }

    return maxSum;
}
```

$O(1)$

$O(n)$ $j - i + 1$

$O(1)$

$O(n^2)$ $\sum_{j=i}^{n-1} (j - i)$

$O(n^3)$ $\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i)$

Triply nested loop: O(n^3)

$O\left(n^3\right)$ time !

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 \text{ iterations}$$

# ordered triplets (i, j, k) $\quad 1 \le i \le k \le j \le n$

$$n(n+1)(n+2)/6 \qquad = (n^3 + 3n^2 + 2)/6$$

$$(n^3 + \cancel{3n^2} + \cancel{2})/\cancel{6} \qquad = O(n^3)$$

Is there another way?

1, 2, -4, 1, 2, -1, 4, -2, 1

How can we do this faster? Notice! Sum(i,j+1)=Sum(i,j)+A(j+1)!

Sum(1,1)=1

Sum(1,2)=1+2=3 Sum(1,2)=Sum(1,1)+2=3

Sum(1,3)=1+2-4=-1 Sum(1,3)=Sum(1,2)-4=-1

⋮

Sum(3,3)=-4

Sum(3,4)=Sum(3,3)+1=-4+1=-3

Sum(3,5)=Sum(3,4)+2=-3+2=-1

⋮

Sum(3,9)=Sum(3,8)+1=0+1=1

Faster Algorithm

- Using Sum(i, j+1) = Sum(i,j) + A[j+1]; remember Sum(i,j) instead of recomputing it.
- Doubly nested for loop: $O(n^2)$

Max Contiguous Subsequence Problem

{-2, 11, -4, 13}

	{}	0
	{-2}	-2
	{-2, 11}	(-2)+11=9
	{-2, 11, -4}	(9)-4= 5
	{-2, 11, -4, 13}	(5)+13=18
	{11}	11
	{11, -4}	(11)-4=7
Maximum	{11, -4, 13}	(7)+13=20
	{-4}	-4
	{-4, 13}	(-4)+13=9
	{13}	13

Algorithm

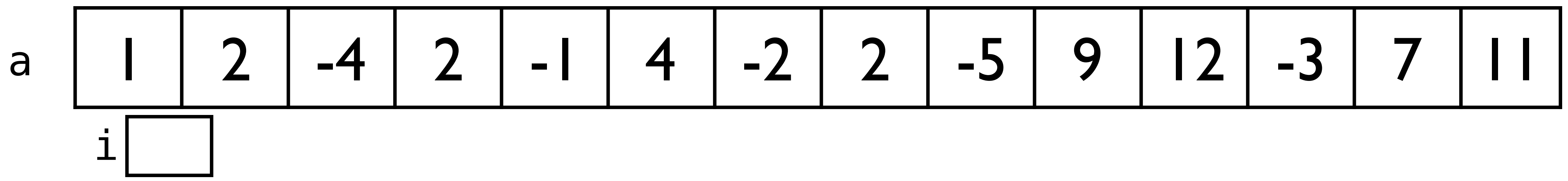
```
int maxSubSum2( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
    {
        int thisSum = 0;
        for( int j = i; j < a.size( ); ++j )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

For each pair i,j
compute the sum of the elements from i to j,
keeping track of the sum(i,j-1) and the largest o
you found, maxsofar.



Algorithm $O(n^2)$ time !

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

    for( int i = 0; i < a.size( ); ++i )
    {
        int thisSum = 0;
        for( int j = i; j < a.size( ); ++j )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

$O(1)$

$O(1)$

$O(1)$

$\sum_{j=i}^{n-1} 1$ $O(n)$

$\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1$ $O(n^2)$

$O\left(n^2\right)$ time !

$$\sum_{i=1}^{n}\sum_{j=i}^{n} 1 \text{ iterations}$$

# ordered pairs (i, j) $\quad 1 \le i \le j \le n$

$$n(n+1)/2 = (n^2+n)/2$$

$$(n^2+\cancel{n})/\cancel{2} = O(n^2)$$

Any other improvements?

- 1, 2, -4, 1, 2, -1, 4, -2, 1

Should we ever check a subsequence starting with A3?
Or A6? Or A8?

Should we ever check a subsequence ending at A4 starting with A1?
Or A2? Or A3?

Should we ever check a subsequence ending at A5 starting with A1?
Or A2? Or A3?

The moral

Any substring that has a maximal value does not contain a prefix which is negative!

and any positive prefix is better than no prefix!

Largest Sum Substring Algorithm

1, 2, -4, 1, 2, -1, 4, -2, 1

	Thissum=0	Maxsum=0
1	Thissum=1	Maxsum=1
1+2=3	Thissum=3	Maxsum=3
3-4=-1	Thissum=0	Maxsum=3
0+1=1	Thissum=1	Maxsum=3
1+2=3	Thissum=3	Maxsum=3
3-1=2	Thissum=2	Maxsum=3
2+4=6	Thissum=6	Maxsum=6
6-2=4	Thissum=4	Maxsum=6
4+1=5	Thissum=5	Maxsum=6

```
template <class Comparable>
Comparable maxSubsequenceSum4( 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;
}
```

Algorithm

For every i
keeping track of the largest subsequence ending at i-1, and keep track of the largest subsequence (with no restrictions), maxsofar.

a	1	2	-4	2	-1	4	-2	2	-5	9	12	-3	7	11

Algorithm $O(n)$ time !

```
template <class Comparable>
Comparable maxSubsequenceSum4( 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;
}
```

$O(1)$

$\sum_{j=0}^{n-1} 1$ $O(n)$

$O(n)$ time !

$$\sum_{j=1}^{n} 1 \text{ iterations}$$

$$1 \leq j \leq n$$

$$n = O(n)$$