

§4 Shellsort ---- by Donald Shell

[[Example] Sort

:

81	94	11	96	12	35	17	95	28	58	41	75	15
----	----	----	----	----	----	----	----	----	----	----	----	----

5-sort

35	17	11	28	12	41	75	15	96	58	81	94	95
----	----	----	----	----	----	----	----	----	----	----	----	----


3-sort

28	12	11	35	15	41	58	17	94	75	81	96	95
----	----	----	----	----	----	----	----	----	----	----	----	----

1-sort

11	12	15	17	28	35	41	58	75	81	94	95	96
----	----	----	----	----	----	----	----	----	----	----	----	----

 Define an *increment sequence* $h_1 < h_2 < \dots < h_t$ ($h_1 = 1$)

 Define an h_k -sort at each phase for $k = t, t - 1, \dots, 1$

Note: An h_k -sorted file that is then h_{k-1} -sorted **remains** h_k -sorted.

☞ Shell's increment sequence:

$$h_t = \lfloor N / 2 \rfloor, \quad h_k = \lfloor h_{k+1} / 2 \rfloor$$

```

void Shellsort( ElementType A[ ], int N )
{
    int i, j, Increment;
    ElementType Tmp;
    for ( Increment = N / 2; Increment > 0; Increment /= 2 )
        /*h sequence */
        for ( i = Increment; i < N; i++ ) { /* insertion sort */
            Tmp = A[ i ];
            for ( j = i; j >= Increment; j -= Increment )
                if( Tmp < A[ j - Increment ] )
                    A[ j ] = A[ j - Increment ];
                else
                    break;
            A[ j ] = Tmp;
        } /* end for-i and for-Increment loops */
}

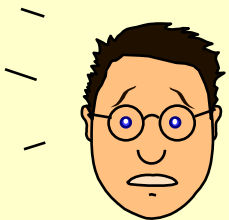
```

👉 Worst-Case Analysis:

【 Theorem 】 The worst-case running time of Shellsort, using Shell's increments, is $\Theta(N^2)$.

[[Example]] A bad case:

	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
8-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
4-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
2-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
1-sort	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

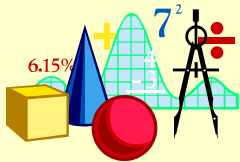


Pairs of increments are not necessarily relatively **prime**.
Thus the smaller increment can have little effect.

☞ Hibbard's Increment Sequence:

$h_k = 2^k - 1$ ---- consecutive increments have no common factors.

【 Theorem 】 The worst-case running time of Shellsort, using Hibbard's increments, is $\Theta (N^{3/2})$.



Conjectures:

$$\text{☞ } T_{\text{avg} - \text{Hibbard}} (N) = O (N^{5/4})$$

Shellsort is a very simple algorithm, yet with an extremely complex analysis. It is good for sorting up to moderately large input (tens of thousands).

☞ Sedgewick's best sequence is $\{1, 5, 19, 41, 109, \dots\}$ in which the terms are either of the form $9 \times 4^i - 9 \times 2^i + 1$ or $4^i - 3 \times 2^i + 1$. $T_{\text{avg}} (N) = O (N^{7/6})$ and $T_{\text{worst}} (N) = O (N^{4/3})$.

§5 Heapsort

Algorithm 1:

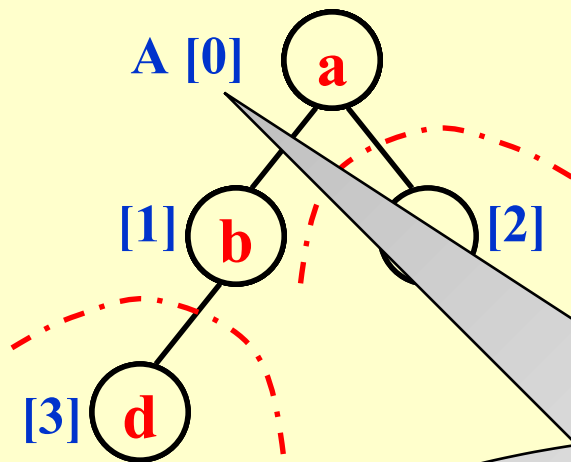
```
{  
    BuildHeap( H );    /* O( N ) */  
    for ( i=0; i<N; i++ )  
        TmpH[ i ] = DeleteMin( H );    /* O( log N ) */  
    for ( i=0; i<N; i++ )  
        H[ i ] = TmpH[ i ];    /* O( 1 ) */  
}
```

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



The space requirement is doubled.

Algorithm 2:



```

void Heapsort( ElementType A[ ], int N )
{
    int i;
    for ( i = N / 2; i >= 0; i -- ) /* BuildHeap */
        PercDown( A, i, N );
    for ( i = N - 1; i > 0; i -- ) {
        Swap( &A[ 0 ], &A[ i ] ); /* DeleteMax */
        PercDown( A, 0, i );
    }
}

```

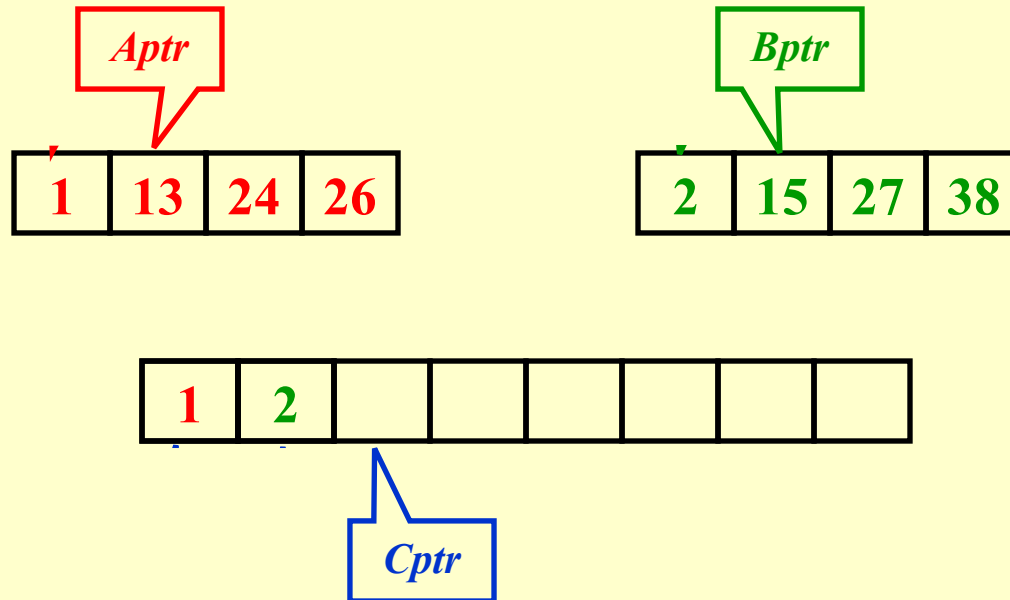
Heapsort data start from

【 Theorem 】 The average number of comparisons used to heapsort a random permutation of N distinct items is $2N \log N - O(N \log \log N)$.

Note: Although Heapsort gives the **best average time**, in practice it is slower than a version of Shellsort that uses Sedgewick's increment sequence.

§6 Mergesort

1. Merge two sorted lists



$T(N) = O(N)$ where N is the total number of elements.

2. Mergesort

```

void MSort( ElementType A[ ], ElementType TmpArray[ ],
           int Left, int Right )
{
    int Center;
    if ( Left < Right ) { /* if there are elements to be sorted */
        Center = ( Left + Right ) / 2;
        MSort( A, TmpArray, Left, Center );      /* T( N / 2 ) */
        MSort( A, TmpArray, Center + 1, Right ); /* T( N / 2 ) */
        Merge( A, TmpArray, Left, Center + 1, Right ); /* O( N ) */
    }
}

```

```

void Merge
{
    Elem
    Tmp
    if (

```

If a TmpArray is declared locally for each call of Merge, then

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

```

    free
}

```

```

else FatalError( "No space for tmp array!!!" );
}

```



```
/* Lpos = start of left half, Rpos = start of right half */
void Merge( ElementType A[ ], ElementType TmpArray[ ],
            int Lpos, int Rpos, int RightEnd )
{
    int i, LeftEnd, NumElements, TmpPos;
    LeftEnd = Rpos - 1;
    TmpPos = Lpos;
    NumElements = RightEnd - Lpos + 1;
    while( Lpos <= LeftEnd && Rpos <= RightEnd ) /* main loop */
        if ( A[ Lpos ] <= A[ Rpos ] )
            TmpArray[ TmpPos++ ] = A[ Lpos++ ];
        else
            TmpArray[ TmpPos++ ] = A[ Rpos++ ];
    while( Lpos <= LeftEnd ) /* Copy rest of first half */
        TmpArray[ TmpPos++ ] = A[ Lpos++ ];
    while( Rpos <= RightEnd ) /* Copy rest of second half */
        TmpArray[ TmpPos++ ] = A[ Rpos++ ];
    for( i = 0; i < NumElements; i++, RightEnd - - )
        /* Copy TmpArray back */
        A[ RightEnd ] = TmpArray[ RightEnd ];
}
```

3. Analysis

$$T(1) = 1$$

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

$$= 2^k T(N/2^k) + k * O(N)$$

$$= N * T(1) + \log N * O(N)$$

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

Note: Mergesort requires **linear extra memory**, and copying an array is slow. It is hardly ever used for internal sorting, but is quite useful for **external sorting**.

Iterative version :

