Chapter 2

Divide-and-Conquer

큰 문제는
작은 문제들로 나누어서
각기 해결한 후에
결과들을 종합하여
전체 답을 얻는다.

문제의 크기가 충분히 작아질 때까지
분할을 계속한다.

Binary Search

- Binary Search (**Algorithm 2.1**)

  locationout = location(1, n);

  input: *keytype* x, *keytype* S[1..n]

```
index location(index low, index high)
{
   iindex mid;
   if (low > high) return 0;     // not found
   mid = (low + high) / 2;
   if (x == S[mid]) return mid;
   else if (x < S[mid])
         return location(low, mid-1);
   else
         return location(mid+1, high);
}
```

- Worst-Case Time Complexity
  - basic operation : comparison of x with S[mid]
  - input size : n (assume $n = 2^k$ for simplicity)
  - recurrence relation

    $W(1) = 1$

    $W(n) = W(n/2) + 1 = W(n/4) + 2 = W(n/8) + 3$

    $= \ldots = W(1) + \lg n = 1 + \lg n \in \Theta(\lg n)$
  - if n is not a power of 2,

    $W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$
- Best-Case: B(n) = ???
- Average-Case: A(n) = ???
- T(n) = ???

- Iterative Form of Binary Search

```
locationout = location2(n);
input: keytype x, keytype S[1..n]
index location2(int n)
{ index mid, low, high;
  low = 1, high = n;
  while (1) {
        if (low > high) return 0;      // not found
        mid = (low + high) / 2;
        if (x == S[mid]) return mid;
        else if (x < S[mid])
              high = mid-1;
        else
              low = mid+1;
  }
}
```

- Time Complexity는 location()과 동일

- Transform into Iterative Form

- efficient if no operations are done after the recursive call (***tail-recursion***)
  - ⇐ stack memory can be eliminated
  - ⇒ faster in a constant factor

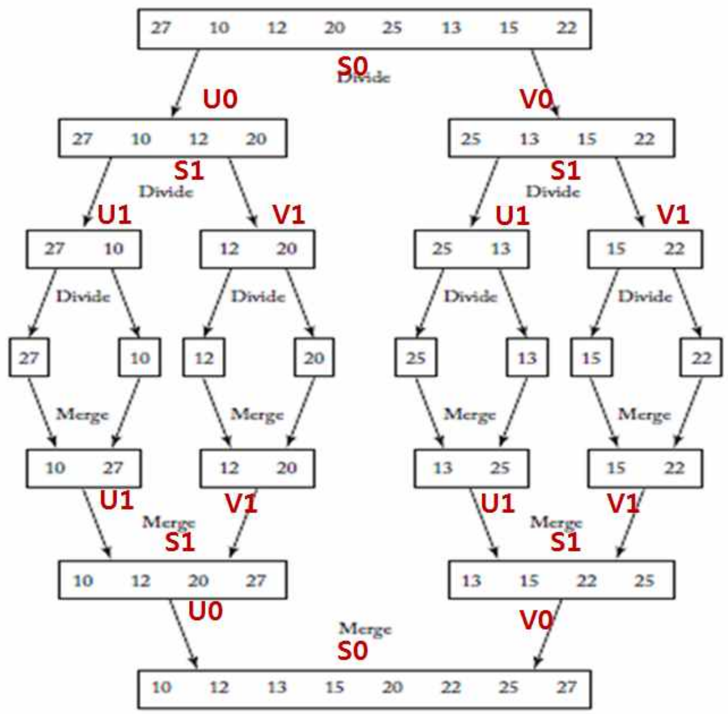
Mergesort

- Mergesort

input: *keytype* S[1..n]

p.59 **Algorithm 2.2 mergesort** and Fig. 2.2

p.59 **Algorithm 2.3 merge** and Table 2.1



● Algorithm 2.2 mergesort

input: *keytype* S[1..n]

```
void mergesort(int n, keytype S[])
{   if (n > 1) {
        index h = n/2; m = n – h;
        keytype U[1..h], V[1..m];

        copy S[1..h] to U[1..h];
        copy S[h+1 .. n] to V[1..m];
        mergesort(h, U);
        mergesort(m, V);
        merge(h, m, U, V, S);
    }
}
```

● Algorithm 2.3 merge

```
void merge(index h, index m,
           keytype U[], keytype V[], keytype S[])
{   index i, j, k;
    i = 1, j = 1, k = 1;
    while (i <= h && j <= m) {
        if (U[i] < V[j]) { S[k] = U[i], i++; }
        else             { S[k] = V[j], j++; }
        k++;
    }
    if (i > h)
        copy V[j] through V[m] to S[k] through S[h+m];
    else
        copy U[i] through U[h] to S[k] through S[h+m];
}
```


● Worst-Case Time Complexity of **merge**

- basic operation : comparison, (or assignment)
- input size : h and m
- $W(h,m) = h+m-1$ or $W(n) = n-1$

? Best-Case

● Worst-Case Time Complexity of **mergesort**

- basic operation : comparison, (or assignment)
- input size : n (assume $n = 2^k$ for simplicity)
- recurrence relation

$$
\begin{aligned}
W(1) &= 0 \\
\mathbf{W(n)} &= W(h) + W(m) + h + m - 1 \\
&= \mathbf{2W(n/2) + [n-1]} \\
&= 2[2W(n/4) + n/2 - 1] + [n-1] \\
&= \mathbf{4W(n/4) + [2n-3]} \\
&= 4[2W(n/8) + n/4 - 1] + [2n-3] \\
&= \mathbf{8W(n/8) + [3n-7]} \\
&= \ldots \\
&= \mathbf{nW(n/n) + [(\lg n)n - (n-1)]} \\
&= n \lg n - (n-1) \in \Theta(n \lg n)
\end{aligned}
$$

$\therefore T(n) \in$???

● Best-Case Time Complexity of **mergesort**

B(n) = ???

● Time Complexity of **mergesort**

$B(n) \le T(n) \le W(n)$

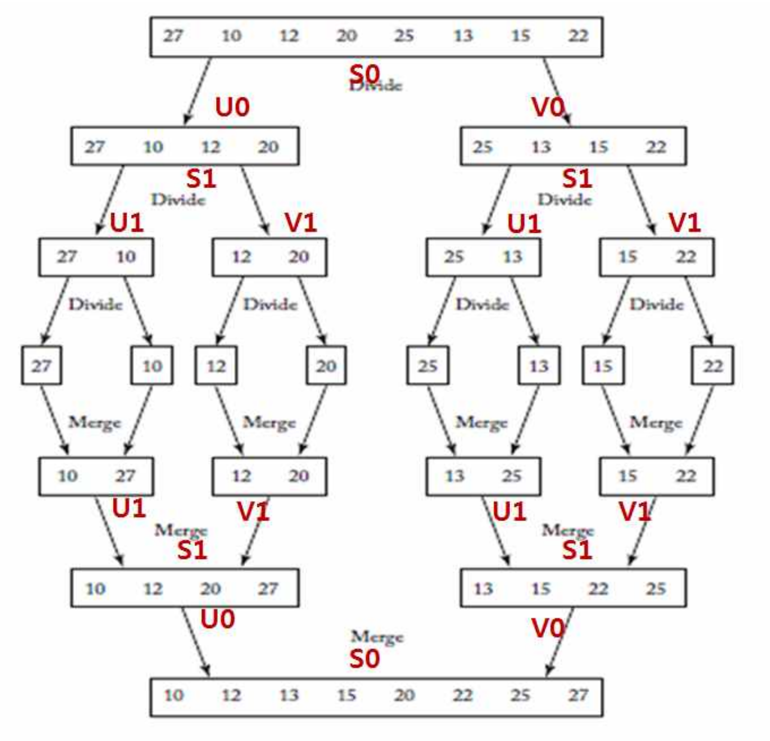
T(n) = ???

● Time Complexity for Assignments ???

Space Complexity of Mergesort

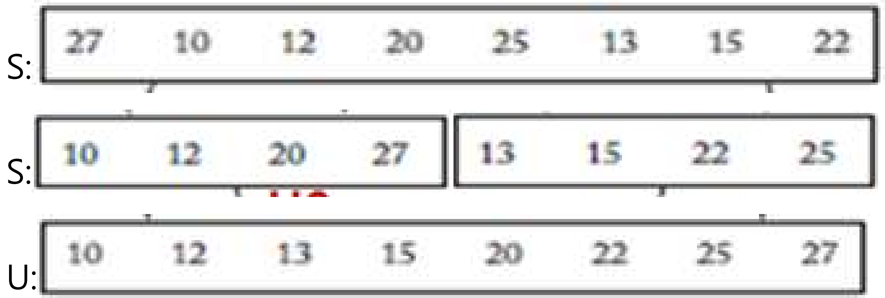
● Space Complexity of **mergesort**
(extra memory space for merge)

- total lg n recursive call
- S(n) = 2 x n/2 + 2 x n/4 + … 2 x 1
  = **2(n-1)** ∈ Θ(n)



● Reduce Extra Space (in-place sort)

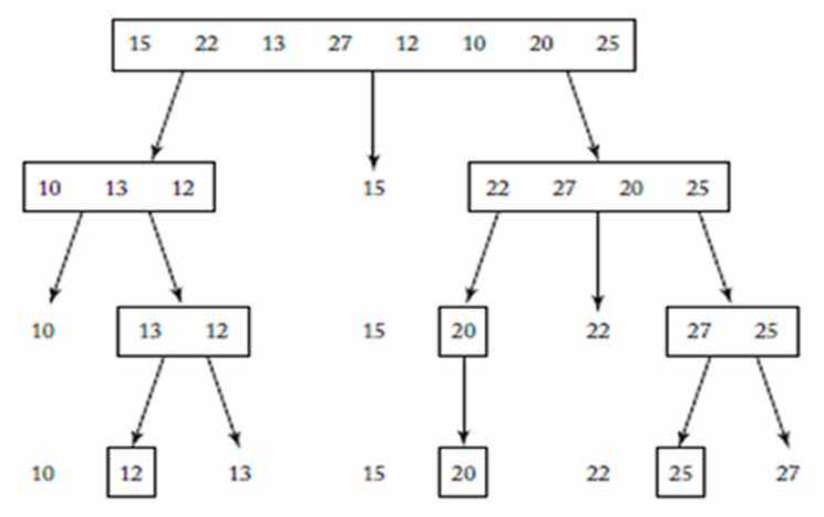
- p.62 **Algorithm 2.4 mergesort2** and **merge2**
  input: S[1..n]
  output: S[1..n]
  첫 호출: mergesort2(1, n)
- S(n) = **n** ∈ Θ(n)



Quicksort

● Quicksort (Partition Exchange Sort)

- the sub-problem instance size is variable



강원대학교 컴퓨터공학과 김용석 (yskim@kangwon.ac.kr)

● Algorithm 2.6 quicksort

input: S[1..n]

output: S[1..n]

첫 호출: quicksort(1, n)

```
void quicksort(index low, index high)
{
    index pivotpoint;
    if (high > low) {
        pivotpoint = partition(low, high);
        quicksort(low, pivotpoint - 1);
        quicksort(pivotpoint + 1, high);
    }
}
```

● Algorithm 2.7 partition

```
index partition(index low, index high)
{
    index I, j, pivotpoint;   keytype pivotitem;
    pivotitem = S[low];
    j = low;                  // j+1과 i 사이는 큰 값들
    for (i=low+1; j <= high; i++)
        if (S[i] < pivotitem) {
            j++;
            exchange S[i] and S[j];
        }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];
    return pivotpoint;
}
```

● 이해하기 쉬운 partition 버전

```
index partition(index low, index high)
{    index i, j, pivotpoint;   keytype pivotitem;

     pivotitem = S[low];
     i = low + 1; j = high;
     while (i<j) {
          while (i<j && S[i] < pivotitem) i++;
          while (i<j && S[j] > pivotitem) j--;
          if (i<j) {
               exchange S[i] and S[j];
               i++, j--;
          }
     }
     pivotpoint = i-1;
     exchange S[low] and S[pivotpoint];
     return pivotpoint;
}
```

● partition 실행 예

i	j	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]	비고
2	8	**15**	22	13	27	12	10	20	25	초기값
2	6	**15**	**22**	13	27	12	**10**	20	25	
3	5	**15**	**10**	13	27	12	**22**	20	25	교환
4	5	**15**	10	13	**27**	**12**	22	20	25	
5	4	**15**	10	13	**12**	**27**	22	20	25	교환
5	4	**15**	10	13	**12**	27	22	20	25	
5	4	**12**	10	13	**15**	27	22	20	25	교환

● Time Complexity of **partition**

- Basic operation : comparison
- $T(n) = n-1$

● Worst-Case Time Complexity of **quicksort**

- $W(n) = W(0) + W(n-1) + (n-1)$
  $= W(n-1) + (n-1)$
  $= W(n-2) + (n-2) + (n-1)$
  $= \ldots$
  $= 1 + \ldots + (n-2) + (n-1)$
  $= n(n-1)/2 \in \Theta(n^2)$

● Time Complexity of **quicksort**

- $T(n) \leq W(n)$
- $T(n) \in$???

● Worst-Case for Exchanges ???
(same as Comparisons)

● Average-Case for Comparisons

- $A(n) = 1.38(n+1) \lg n \in \Theta(n \lg n)$
- Merge sort: $n \lg n - (n-1) \in \Theta(n \lg n)$

● Average-Case for Assignments (p.315 Tab. 7.2)

- Quick sort: $0.69\ n \lg n \in \Theta(n \lg n)$
- Merge sort: $2\ n \lg n \in \Theta(n \lg n)$

Divide-and-Conquer

● General form of Divide-and-Conquer algorithm

```
DAC(P) {
   if Small(P) {              // terminal condition
      return Solution(P);
   } else {
      divide P into P1, P2, ..., Pk
      return Combine(DAC(P1), ..., DAC(Pk))
} }
```

● Time complexity

$T(n) = g(n)$, *if n is small*

$T(n_1) + \dots + T(n_k) + f(n)$, *otherwise*

$f(n)$: time complexity of Combine

Generally, $T(n) = aT(n/b) + f(n)$ for $n>1$

Assume $n = b^h$ for simplicity

(참고) Time Complexity 연습

● 연습: Stooge Sort (see Wikipedia)

```
keytype S[1..n];
void StoogeSort(int low, int high)
{
   if (high - low < 2) {
      if (S[low] > S[high])
         exchange(S[low], S[high]);
   } else {
      k = □(high - low + 1) / 3□;
      StoogeSort(low, high - k);
      StoogeSort(low + k, high);
      StoogeSort(low, high - k);
   }
}
```

- Divide and conquer ???
- Correctness: ???
- $T(n) = ???$
- $T(n) \in O(n^{\log 3/\log 1.5}) \approx O(n^{2.71})$

Large Integer Arithmetic

- Representation of Large Integers
  - array of small (0-9) integers
  - one slot for sign
- Linear Time Addition

```
// assume non-negative large integers
add (int n, large_int u, large_int v, large_int &s) {
      int i, carry=0;
      for (i=0; i < n; i++) {
            s[i] = u[i] + v[i] + carry;
            if (s[i] >= 10) {
                  s[i] -= 10;
                  carry = 1;
            } else {
                  carry = 0;
            }
}     }
```

- basic operations

- add/subtract/compare integers
- $T(n) \in \Theta(n)$
- ? in the case of 32bit processors

- More Linear Time Operations
  - $u \times 10^m$
  - $u / 10^m$
  - $u \% 10^m$

Large Integer Multiplication

- Large Integer Multiplication of n digits
  - $u = x \cdot 10^m + y$, $v = w \cdot 10^m + z$,
    where $m = \lfloor n/2 \rfloor$, y and z has m digits,
    and x and w has n-m digits
  - $uv = xw \cdot 10^{2m} + (xz+yw) \cdot 10^m + yz$
- Divide-and-Conquer Algorithm (p.78 Alg. 2.9)

```
large_int prod (large_int u, large_int v) {
   n = max(# of digits in u, # of digits in v);
   if (u == 0 || v == 0) {
      return 0;
   } else if (n <= threshold) {
      return u*v;
   } else {
      m = ⌊n/2⌋; // m = n / 2
      x = u / 10^m; y = u % 10^m;
      w = v / 10^m; z = v % 10^m;
      return prod(x,w)*10^2m
             + (prod(x,z)+prod(w,y))*10^m + prod(y,z);
} }
```

- Worst-Case Time Complexity

- basic operations : 1 digit add, 1 digit multiply,
- $W(n) = 4W(n/2) + cn$, for $n > s$
  d for $n <= s$ (*assume* $s=1$)
- $W(n) = 4[4W(n/4) + cn/2] + cn = 4^2 W(n/4) + 3cn$
  $= \ldots = 4^k W(n/n) + (n-1)cn = (d+c)n^2 - cn$

$\therefore W(n) \in \Theta(n^2)$

- Enhancement of Time Complexity
  - need 4 multiplications : xw, xz+yw, yz
  - r = (x+y)(w+z) = xw + (xz+yw) + yz
    i.e., xz + yw = r - xw - yz
    Then, we need 3 multiplications for xw, yz, r

```
large_int prod2 (large_int u, large_int v) {
  n = max(# of digits in u, # of digits in v);
  if (u == 0 || v == 0) {
      return 0;
  } else if (n <= threshold) {
      return u*v;
  } else {
      m = ⌊n/2⌋;    x = u / 10^m; y = u % 10^m;
      w = v / 10^m; z = v % 10^m;
      p = prod2(x,w); q = prod2(y,z);
      r = prod2(x+y, w+z);
      return p*10^2m + (r-p-q)*10^m + q;
} }
```

- Worst-Case Time Complexity
  - basic operations : add, divide 10^m, rem 10^m
  - $W(n) = 3W(n/2) + cn$, for $n > s$
    d for $n <= s$
  - $W(n) = 3[3W(n/4)+cn/2]+cn = 3^2W(n/4)+(1+3/2)cn$
    $= \ldots$
    $= 3^kW(n/n) + [1+(3/2)^1 + \ldots + (3/2)^{k-1}]cn$
    $= d\cdot 3^k + 2\cdot[(3/2)^k - 1]\cdot cn$
  $\therefore W(n) \in \Theta(n^{\lg 3} + n^{1+\lg 3/2}) = \Theta(n^{\lg 3}) \approx \Theta(n^{1.58})$

More Improvements

- Borodin and Munro (1975) : $\Theta(n(\lg n)^2)$
  - may be the same for divide, square root, ...

Matrix Multiplication

- Multiplication of n by n matrices : C = A x B

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)B(k,j)$$

- Number of Multiplications : n x n^2 = n^3
- Number of Additions : (n-1) x n^2 = n^3-n^2

- Divide into 4 n/2 by n/2 matrices

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

where
$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$


```
● MatrixMul( int n, matrix A, matrix B, matrix& C)
{   matrix U, V;
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0]; return;
    }
    divide A into submatrices A11, A12, A21, A22;
    divide B into submatrices B11, B12, B21, B22;
    MatrixMul(n/2, A11, B11, U);
    MatrixMul(n/2, A12, B21, V);
    C11 = U + V;        // add(n/2, U, V, C11);
    MatrixMul(n/2, A11, B12, U);
    MatrixMul(n/2, A12, B22, V);
    C12 = U + V;        // add(n/2, U, V, C12);
    MatrixMul(n/2, A21, B11, U);
    MatrixMul(n/2, A22, B21, V);
    C21 = U + V;        // add(n/2, U, V, C21);
    MatrixMul(n/2, A21, B12, U);
    MatrixMul(n/2, A22, B22, V);
    C22 = U + V;        // add(n/2, U, V, C22);

    combine C11, C12, C21, C22 into C;
}
```

Note) Assume n = 2^k

(odd n: divide into ⌊n/2⌋ and ⌈n/2⌉)

- Number of Multiplications

$$
\begin{aligned}
T(1) &= 1,\\
T(n) &= 8T(n/2), \text{ for } n>1\\
&= 8^2T(n/4) = 8^3T(n/8) = \dots = 8^kT(1)\\
&= n^3 \in \Theta(n^3)
\end{aligned}
$$

- Number of Additions

$$
\begin{aligned}
T(1) &= 0,\\
T(n) &= 8T(n/2) + 4(n/2)^2, \text{ for } n>1\\
&= 8[8T(n/4)+4(n/4)^2] + n^2 = 8^2T(n/4)+3n^2\\
&= 8^2[8T(n/8)+4(n/8)^2] + 3n^2 = 8^3T(n/8)+7n^2\\
&\dots\\
&= 8^kT(1) + (n-1)n^2 = n^3-n^2 \in \Theta(n^3)
\end{aligned}
$$

Strassen's Matrix Multiplication

$$
\begin{aligned}
M_1 &= (A_{11}+A_{22})(B_{11}+B_{22})\\
M_2 &= (A_{21}+A_{22})B_{11}\\
M_3 &= A_{11}(B_{12}-B_{22})\\
M_4 &= A_{22}(B_{21}-B_{11})\\
M_5 &= (A_{11}+A_{12})B_{22}\\
M_6 &= (A_{21}-A_{11})(B_{11}+B_{12})\\
M_7 &= (A_{12}-A_{22})(B_{21}+B_{22})
\end{aligned}
$$

⇨ $C_{11} = M_1+M_4-M_5+M_7$ $C_{12} = M_3+M_5$

$C_{21} = M_2+M_4$ $C_{22} = M_1+M_3-M_2+M_6$

- The Algorithm : p.74 **Algorithm 2.8 strassen**

- Number of Multiplications

$$
\begin{aligned}
T(1) &= 1,\\
T(n) &= 7T(n/2), \text{ for } n>1\\
&= 7^2T(n/4) = 7^3T(n/8) = \dots = 7^kT(1)\\
&= 7^{\lg n} = n^{\lg 7} \approx n^{2.807} \in \Theta(n^{2.807})
\end{aligned}
$$

- Number of Additions

$$
\begin{aligned}
T(1) &= 0,\\
T(n) &= 7T(n/2) + 18(n/2)^2, \text{ for } n>1\\
&= 7^2T(n/4) + 7\text{x}18(n/4)^2 + 18(n/2)^2\\
&\dots\dots\\
&= 6n^{\lg 7} - 6n^2 \approx 6n^{2.807} - 6n^2 \in \Theta(n^{2.807})
\end{aligned}
$$

김용석 (yskim@kangwon.ac.kr)

- **strassen**(int n, matrix A, matrix B, matrix& C)

```
{    matrix M1, M2, ..., M7, U, V;
     if (n == 1) {
          C[0][0] = A[0][0] * B[0][0]; return;
     }
     divide A into submatrices A11, A12, A21, A22;
     divide B into submatrices B11, B12, B21, B22;
     strassen(n/2, A11+A22, B11+B22, M1);
     // add(n/2, A11, A22, U); add(n/2, B11, B22, V);
     // strassen(n/2, U, V, M1);
     strassen(n/2, A21+A22, B11, M2);
     ...
     strassen(n/2, A12-A22, B21+B22, M7);
     C11 = M1 + M4 - M5 + M7
     // add(n/2, M1, M4, U); sub(n/2, U, M5, V);
     // add(n/2, V, M7, C11);
     C12 = M3 + M5;      // add(n/2, M3, M5, C12);
     C21 = M2 + M4;      // add(n/2, M2, M4, C21);
     C22 = M1 + M3 - M2 + M6
     // add(n/2, M1, M3, U); sub(n/2, U, M2, V);
     // add(n/2, V, M6, C22);
     combine C11, C12, C21, C22 into C;
}
```


김용석 (yskim@kangwon.ac.kr)

More Improvements

- Number of Additions
  $5n^{2.807} - 5n^2$ by S. Winograd, 1980
- Number of Multiplications
  1978 $O(n^{2.788})$,
  1979 $O(n^{2.779})$, $O(n^{2.522})$,
  1982 $O(n^{2.495})$,
  1987 $O(n^{2.376})$ by Coppersmith and Winograd
  - practically, the constant is too large
  ⇨ Strassen's algorithm is more efficient
- Lower bound of the time complexity
  Proved as $\Omega(n^2)$
- Open Problem
  Exist any algorithm of $O(n^2)$?

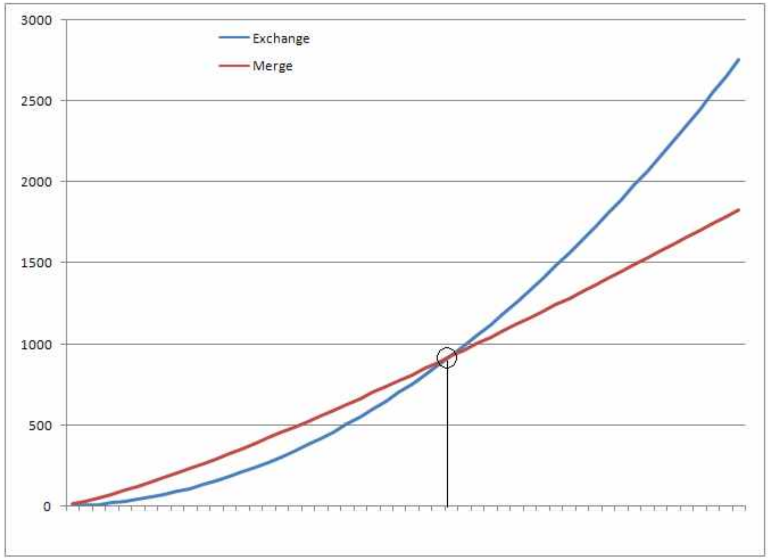
Determining Thresholds

● Sorting Example (Ex. 2.7 p.85)

Assume that

$W_{merge}(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + 32n$ us

$W_{exchange}(n) = n(n-1)/2$ us



● Select the threshold (largest) ***t*** such that

$W_{exchange}(n) \leqq W_{merge}(n)$ for $n \leqq t$

Then, we use exchange-sort for $n \leqq t$,

and merge-sort, otherwise

Therefore,

$W(n) = n(n-1)/2$ for $n \leqq t$

$W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + 32n$ for $n > t$

To determine the largest t such that

$W(\lfloor t/2 \rfloor) + W(\lceil t/2 \rceil) + 32t \geqq t(t-1)/2$

Even t :

$2 * (t/2)(t/2 - 1)/2 + 32t \geqq t(t-1)/2$

$t^2 - 2t + 128t < 2t^2 - 2t$

$t^2 - 128t = t(t-128) \leqq 0$

therefore, $t \leqq 128$

Odd t :

$[(t-1)/2][(t-1)/2-1]/2 + [(t+1)/2][(t+1)/2-1]/2$

$+ 32t$

$= 1/4(t^2 - 2t + 1) + 32t \geqq t(t-1)/2$

$t^2 - 2t + 1 + 128t < 2t^2 - 2t$

$t^2 - 128t - 1 \leqq 0$

therefore, $t \leqq 128.008$

⇒ The threshold (the largest integer) ***t*** is 128