Chapter 2. (Part 1)

Divide-and-Conquer

경북대학교 배준현 교수

(joonion@knu.ac.kr)

Contents

- The **Divide-and-Conquer** Approach
  - *divides* an instance of a problem into *two or more smaller instances*.
    - The divided smaller instances are also instances of the problem.
    - If they are *still too large* to be solved readily,
      - they can be divided into *still smaller instances*.
    - If solutions to them can be obtained readily,
      - these smaller solutions can be *combined* into the original solution.
  - It is a ***top-down approach***, that is,
    - the solution to a *top-level instance* of a problem is obtained
    - by *going down* and *obtaining solutions* to smaller instances.

2.1 Binary Search

- The steps of Binary Search:
  - If x equals the middle item, then quit. Otherwise:

1. ***Divide*** the array into *two subarrays* about half as large.
   - If x is *smaller* than the *middle* item, choose the *left* subarray.
   - If x is *larger* than the *middle* item, choose the *right* subarray.
2. ***Conquer*** (solve) the subarray
   - by determining whether x is in that subarray.
   - Unless the subarray is sufficiently small, use *recursion* to do this.
3. *Obtain* the solution to the array from the solution to the subarray.

2.1 Binary Search

$x = 18$

$S =$

10	12	13	14	18	20	**25**	27	30	35	40	45	47

Choose left subarray because $x < 25$

Compare x with 25

10	12	**13**	14	18	20

Compare x with 13

Choose right subarray because $x > 13$

14	**18**	20

Compare x with 18 → Determine that x is present because $x = 18$

2.1 Binary Search

ALGORITHM 2.1: Binary Search (Recursive)

```
int binsearch2(int low, int high) {
    int mid;

    if (low > high)
        return 0;
    else {
        mid = (low + high) / 2;
        if (x == S[mid])
            return mid;
        else if (x < S[mid])
            return binsearch2(low, mid - 1);
        else // x > S[mid]
            return binsearch2(mid + 1, high);
    }
}
```

2.1 Binary Search

- Implementing the *Recursive* Binary Search:
  - Note that n, S, and x *are not parameters* to the function $binsearch2$.
    - Only the variables *whose values can change in the recursive calls*
      - are made parameters to recursive routines.
    - Hence, define n, S, and x as *global variables*.
  - Then, our *top-level call* to the function$binsearch2$ and the output would be:

```
// global variables
int n, x;
vector<int> S;


location = binsearch2(1, n);
```

```
[Input]
13
10 12 13 14 18 20 25 27 30 35 40 45 47
18

[Output]
5
```

2.1 Binary Search

- Time Complexity Analysis (*Worst-Case*)
  - Basic Operation: the *comparison* of x with $S[mid]$.
  - Input Size: n, the *number of items* in the array.
  - Note that the worst-case can occur
    - when x is larger than all items in the list.
  - Assume that n is a power of 2.
    - If $n = 1$, then $W(n) = W(1) = 1$.
    - If $n > 1$, then $W(n) = \underbrace{W\left(\frac{n}{2}\right)}_{\text{Comparisons in recursive call}} + \underbrace{1}_{\text{Comparison at top level}}$

- Time Complexity Analysis (Worst-Case)
  - The recurrence equation:
    - $W(1) = 1$, for $n = 1$,
    - $W(n) = W(n/2) + 1$, for $n > 1$ and n is a power of 2.
  - This recurrence is solved to:
    - $W(n) = \lg n + 1 \in \Theta(\lg n)$. (Refer to Example B.1 in Appendix B)
  - If n is not restricted to being a power of 2, then
    - $W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$. (Refer to Exercise 2.1.4)

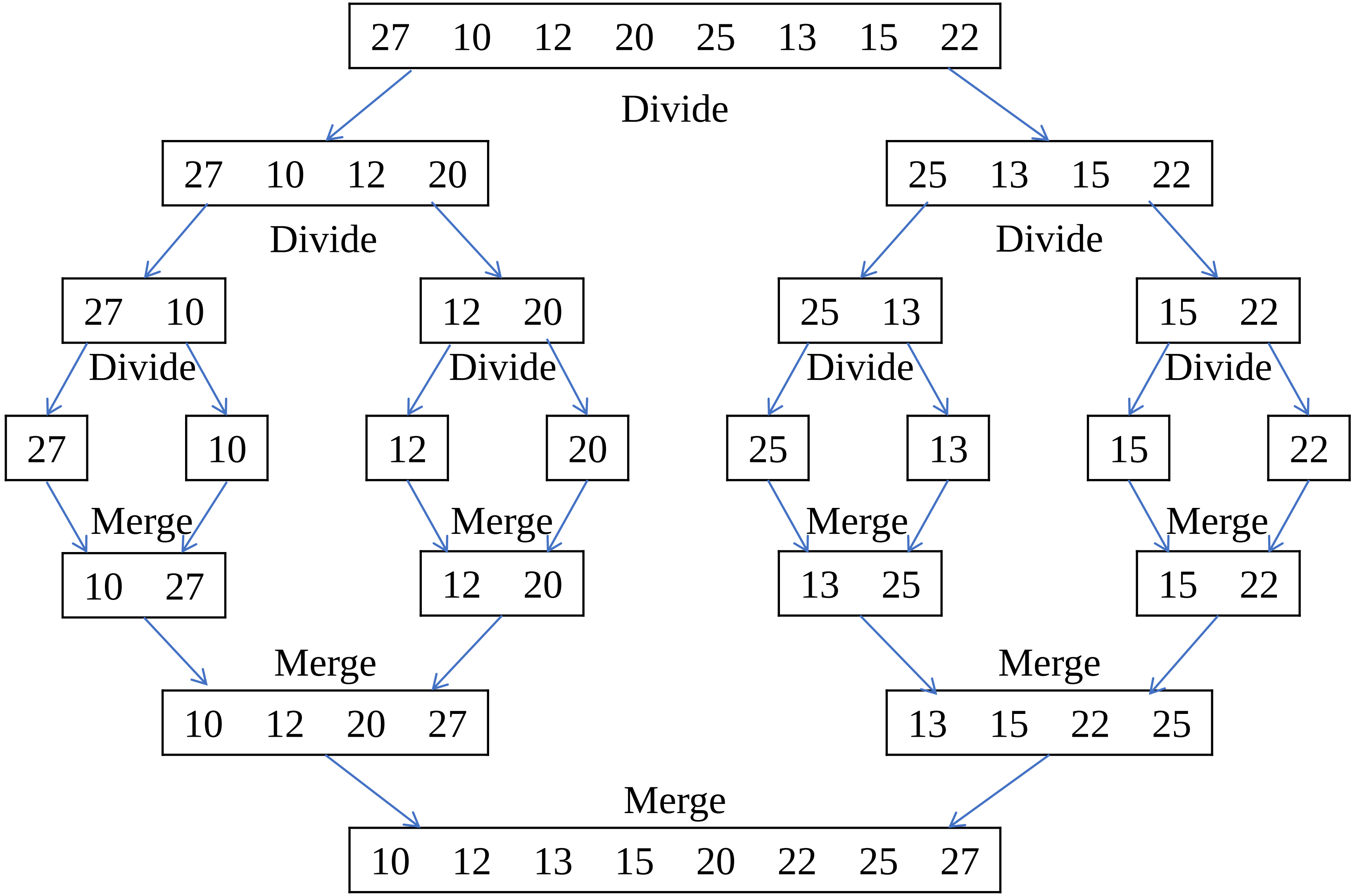
2.2 Mergesort

- Mergesort:
  - *Two-way merging*
    - combines *two sorted arrays* into *one sorted array*.
  - We can sort an array
    - by *repeatedly* apply the *two-way merging* procedure.
  - *Divide* it into two subarrays, *sort* the two arrays, and
    - *merge* them to produce the sorted array.

2.2 Mergesort

- The steps of Mergesort
  1. ***Divide*** the array into two subarrays each with $n/2$ items.
  2. ***Conquer*** (*solve*) each subarray by sorting it.
     - Unless the array is sufficiently small, use *recursion* to do this.
  3. ***Combine*** the solutions to the subarrays
     - by *merging* them into a single sorted array.

2.2 Mergesort



ALGORITHM 2.2: Mergesort

```
void mergesort(int n, vector<int>& S)
{
    if (n > 1) {
        int h = n / 2, m = n - h;
        vector<int> U(h + 1), V(m + 1);
        // copy S[1] through S[h] to U[1] through U[h]
        for (int i = 1; i <= h; i++)
            U[i] = S[i];
        // copy S[h+1] through S[n] to V[1] through V[m]
        for (int i = h + 1; i <= n; i++)
            V[i - h] = S[i];
        mergesort(h, U);
        mergesort(m, V);
        merge(h, m, U, V, S);
    }
}
```

2.2 Mergesort

ALGORITHM 2.3: Merge

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

2.2 Mergesort

- Merging two arrays U and V into one array S.

U	V	S
10 12 20 27	13 15 22 25	10
12 20 27	13 15 22 25	10 12
20 27	13 15 22 25	10 12 13
20 27	15 22 25	10 12 13 15
20 27	22 25	10 12 13 15 20
27	22 25	10 12 13 15 20 22
27	25	10 12 13 15 20 22 25
27		10 12 13 15 20 22 25 27

- **Time Complexity of *Merge* (Worst-Case)**
  - Basic Operation: the *comparison* of $U[i]$ with $V[j]$.
  - Input Size: h and m, the *number of items* in each of the two input arrays.
  - The *worst-case* occurs when the while-loop is exited,
    - one of two indices (i) has reached its exit point ($h + 1$),
    - whereas the other index (j) has reached m (1 less than its exit point).
  - Therefore,
    - $W(h, m) = h + m - 1$.

2.2 Mergesort

- Time Complexity of Mergesort (Worst-Case)
  - Basic Operation: the *comparison* that takes place in $merge$.
  - Input Size: n, the *number of items* in the array S.
  - The total number of comparisons is the sum of
    - the number of comparison in the recursive call to $mergesort$.

$$W(n) = \underset{\substack{\uparrow \\ \text{Time to sort } U}}{W(h)} + \underset{\substack{\uparrow \\ \text{Time to sort } V}}{W(m)} + \underset{\substack{\uparrow \\ \text{Time to merge}}}{h + m - 1}$$

2.2 Mergesort

- Time Complexity of Mergesort (Worst-Case)
  - In the case where n is a power of 2.
    - Establish the recurrence relation:
      - $h = \lfloor n/2 \rfloor = n/2\,, m = n - h = n/2\,, h + m = n.$
      - $W(1) = 0$, for $n = 1$,
      - $W(n) = 2W(n/2) + n - 1$, for $n > 1$, n is a power of 2.
    - Therefore,
      - $W(n) = n \lg n - (n-1) \in \Theta(n \lg n)$ (Example B.19 in Appendix B)
  - In the case where n is *not* a power of 2.
    - $W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1$
    - $W(n) \in \Theta(n \lg n)$ by Theorem B.4 (Example B.25 in Appendix B.4)

2.2 Mergesort

- How about the Space Complexity?
  - An ***in-place sort*** is a sorting algorithm that
    - does not use any *extra space* beyond that needed to store the input.
  - Algorithm 2.2 is *not an in-place sort*,
    - because it uses extra arrays U and V besides the input array S.
  - The total number of extra array items created is about
    - $S(n) = n(1 + \frac{1}{2} + \frac{1}{4} + \cdots) = 2n$
  - It is *possible* to *reduce* the *amount of extra space*
    - to *only one array* containing n items.

Algorithm 2.4: Mergesort 2

```
void mergesort2(int low, int high)
{
    if (low < high) {
        int mid = (low + high) / 2;
        mergesort2(low, mid);
        mergesort2(mid + 1, high);
        merge2(low, mid, high);
    }
}
```

```
// global varaibles
int n;
vector<int> S;
```

```
mergesort2(1, n);
```

ALGORITHM 2.5: Merge 2

```
void merge2(int low, int mid, int high) {
    int i = low, j = mid + 1, k = 0;
    vector<int> U(high - low + 1);

    while (i <= mid && j <= high)
        U[k++] = (S[i] < S[j]) ? S[i++] : S[j++];
    if (i > mid)
        // move S[j] through S[high] to U[k] through U[high]
        while (j <= high)
            U[k++] = S[j++];
    else // j > high
        // move S[i] through S[mid] to U[k] through U[high]
        while (i <= mid)
            U[k++] = S[i++];
    // move U[0] through U[high-low+1] to S[low] through S[high]
    for (int t = low; t <= high; t++)
        S[t] = U[t - low];
}
```

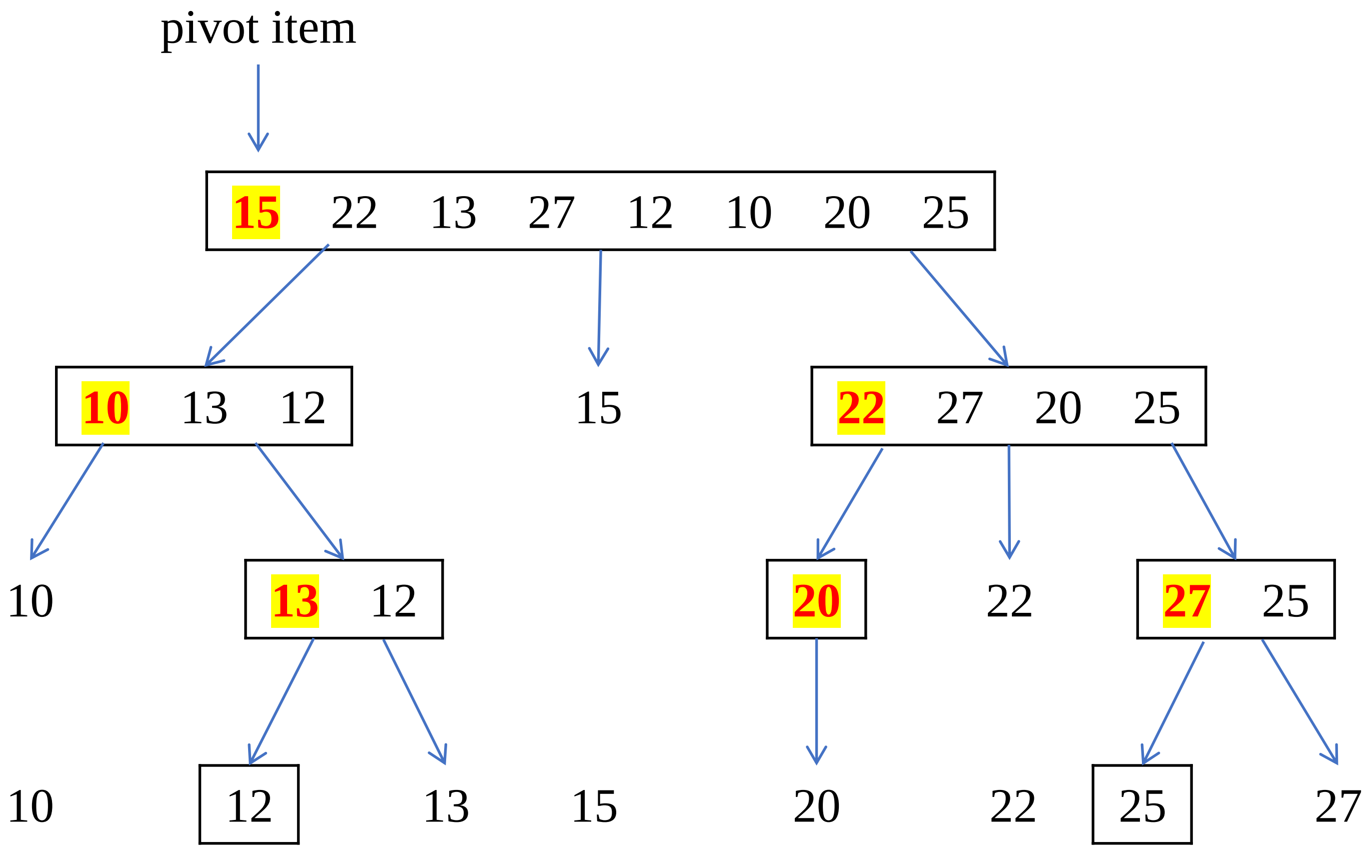
2.3 The Divide-and-Conquer Approach

- The *Design Strategy* of the Divide-and-Conquer:
  1. ***Divide*** an instance of a problem into one or more smaller instances.
  2. ***Conquer*** (*solve*) each of the smaller instances.
     - Unless a smaller instance is sufficiently small, use *recursion* to do this.
  3. *If necessary*, ***combine*** the solutions to the smaller instances
     - to obtain the solution to the original instance.

2.4 Quicksort (Partition Exchange Sort)

- Quicksort
  - is an *in-place* sorting algorithm developed by Hoare (1962).
  - is similar to Mergesort in that
    - it divides the array into *two partitions*
      - and then sorting each partition *recursively*.
  - However, the array is *partitioned*
    - by placing all items *smaller* than some ***pivot item*** *before* that item
      - and all items *larger* than the ***pivot item*** *after* it.
    - the *pivot item* can be *any* item,
      - *for convenience*, we will simply make it the *first one*.

2.4 Quicksort (Partition Exchange Sort)



2.4 Quicksort (Partition Exchange Sort)

ALGORITHM 2.6: Quicksort

```
void quicksort(int low, int high)
{               int r = rand(low, high)
    int pivotpoint; S[r]

    if (low < high) {                                    2개의 파티션으로 나눈다.
        partition(low, high, pivotpoint);
        quicksort(low, pivotpoint - 1);
        quicksort(pivotpoint + 1, high);
    }
}
```

```
// global variables
int n;
vector<int> S;
```

```
quicksort(1, n);
```

2.4 Quicksort (Partition Exchange Sort)

ALGORITHM 2.7: Partition

```
void partition(int low, int high, int& pivotpoint)
{
    int pivotitem = S[low];

    int j = low;
    for (int i = low + 1; i <= high; i++)
        if (S[i] < pivotitem) {
            j++;
            swap(S[i], S[j]);
        }
    pivotpoint = j;
    swap(S[low], S[pivotpoint]);
}
```

2.4 Quicksort (Partition Exchange Sort)

$S[1]$	$S[2]$	$S[3]$	$S[4]$	$S[5]$	$S[6]$	$S[7]$	$S[8]$
15	22	13	27	12	10	20	25
low							*high*

15	**22**	13	27	12	10	20	25
j	i						

15	22	**13**	27	12	10	20	25
j	$j{+}{+}$	i					

15	13	22	**27**	12	10	20	25
	j		i				

15	13	22	27	**12**	10	20	25
	j	$j{+}{+}$		i			

15	13	12	27	22	**10**	20	25
		j	$j{+}{+}$		i		

15	13	12	10	22	27	**20**	25
			j			i	

15	13	12	10	22	27	20	**25**
			j				i

10	13	12	15	22	27	20	25	
low			j					i
			pivotpoint					

2.4 Quicksort (Partition Exchange Sort)

- Time Complexity of *Partition* (Every-Case)
  - Basic Operation: the *comparison* of $S[i]$ with $pivotitem$.
  - Input Size: $n = high - low + 1$, the *number of items* in the subarray.
  - Since every item except the first is compared,
    - $T(n) = n - 1$.

2.4 Quicksort (Partition Exchange Sort)

- Time Complexity of *Quicksort* (Worst-Case)
  - Basic Operation: the *comparison* of $S[i]$ with *pivotitem* in partition.
  - Input Size: n, the *number of items* in the array S.
  - Note that the *worst-case* occurs
    - when the array is *already sorted* in non-decreasing order.
  - If the array is already sorted,
    - *no items are less than* the *first item* (*pivot item*) in the array.
  - Therefore,

$$T(n) = \underbrace{T(0)}_{\text{Time to sort } \textit{left} \text{ subarray}} + \underbrace{T(n-1)}_{\text{Time to sort } \textit{right} \text{ subarray}} + \underbrace{n-1}_{\text{Time to partition}}$$

2.4 Quicksort (Partition Exchange Sort)

- Time Complexity of Quicksort (Worst-Case)
  - recurrence equation:
    - $T(0) = 1$, for $n = 0$,
    - $T(n) \leq \frac{n(n-1)}{2}$, for $n > 0$.
  - the *worst-case* time complexity is:
    - $W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$. (Example B.16 in Appendix B)

2.4 Quicksort (Partition Exchange Sort)

- Time Complexity of *Quicksort* (*Average-Case*)
  - Now assume that the value of *pivotpoint* returned by *partition*
    - is *equally likely* to be *any* of the numbers from 1 through n.
  - In this case, the average-case time complexity is given:

$$A(n) = \sum_{p=1}^{n} \frac{1}{n} \underbrace{[A(p-1) + A(n-p)]}_{\text{Average time to sort subarray when pivotpoint is } p} + n - 1$$

Probability that pivotpoint is p

Time to partition

  - The *approximate* solution to this recurrence is given:
    - $A(n) \approx (n+1)2\ln n = (n+1)2\ln 2\,(\lg n) \approx 1.38(n+1)\lg n \in \Theta(n \lg n)$

- The Analysis of *Recursive Algorithms*:
  - is not as straightforward as it is for *iterative algorithms*.
  - However, it is not difficult to represent
    - the time complexity of a recursive algorithm
    - by a *recurrence equation*.
  - Fortunately, there exist a simple method
    - to solve the recurrence equations with a certain type.
    - called as ***The Master Theorem.***

▪ Theorem B.5 (*Master Theorem*)

- Suppose that a complexity function $T(n)$ satisfies:
  - $T(n) = aT(\frac{n}{b}) + cn^k$, for $n > 1$, n is a power of b,
  - $T(1) = d$, for $n = 1$.
    - where $b \geq 2$ and $k \geq 0$ are constant integers,
    - and a, c, and d are constants such that $a > 0$, $c > 0$, and $d \geq 0$.
- Then,
  - $T(n) \in \Theta(n^k)$, if $a < b^k$.
  - $T(n) \in \Theta(n^k \lg n)$, if $a = b^k$.
  - $T(n) \in \Theta(n^{\log_b a})$, if $a > b^k$.

Appendix B. Solving Recurrence Equations

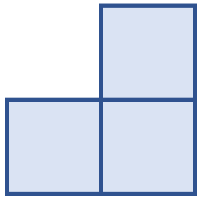
- Examples of Applying the Master Theorem:
  - Example B.26:
    - $T(n) = 8T(n/4) + 5n^2$, for $n > 1$, n is a power of 4.
    - $T(1) = 3$
    - Then, $T(n) \in \Theta(n^2)$, since $a = 8 < b^k = 4^2$.
  - Example B.27:
    - $T(n) = 9T(n/3) + 5n^1$, for $n > 1$, n is a power of 3.
    - $T(1) = 7$
    - Then, $T(n) \in \Theta\left(n^{\log_3 9}\right) = \Theta(n^2)$, since $a = 9 > b^k = 3^1$.

- Examples of Applying the Master Theorem:
  - Example B.28:
    - $T(n) = 8T(n/2) + 5n^3$, for $n > 64$, n is a power of 2.
    - $T(64) = 200$
    - Then, $T(n) \in \Theta(n^3 \lg n)$, since $a = 8 = b^k = 2^3$.
  - The Analysis of the Algorithm 2.2 (*Mergesort*)
    - $W(n) = 2W(n/2) + n - 1$, for $n > 1$, n is a power of 2.
    - $W(1) = 0$
    - Then, $W(n) \in \Theta(n \lg n)$, since $a = 2 = b^k = 2^1$.

Exercises

- Exercise No. 42.
  - A ***tromino*** is a group of three unit squares arranged in an L-shape. Consider the following tiling problem: The input is an $m \times m$ array of unit squares where m is a positive power of 2, with one forbidden square on the array. The output is a tiling of the array that satisfies the following conditions:
    - Every unit square other than the input square is covered by a tromino.
    - No tromino covers the input square.
    - No two trominos overlap.
    - No tromino extends beyond the board.
  - Write a divide-and-conquer algorithm that solves this problem.

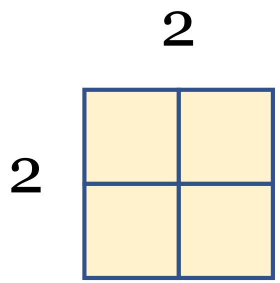
하노이탑 : T(n) = 2 * T(n-1) + 1

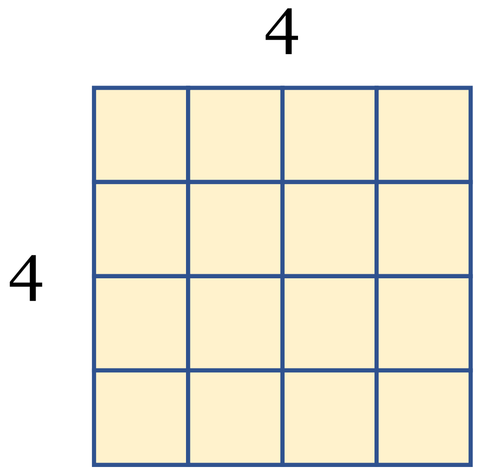


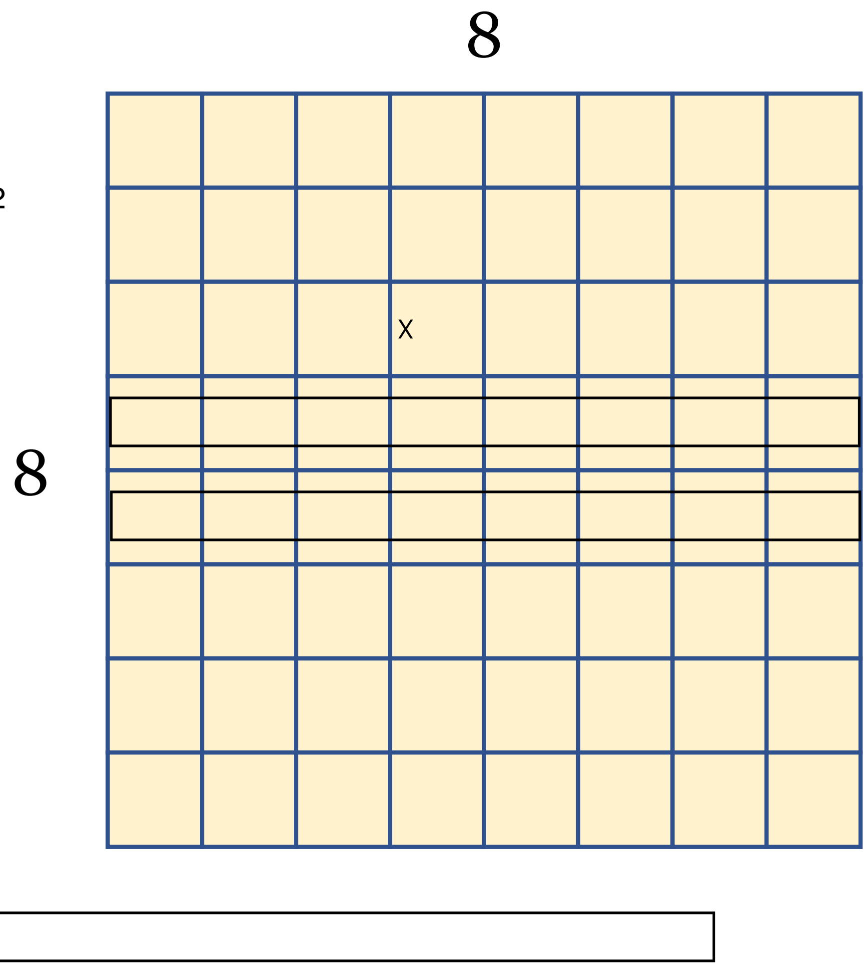
tromino

2차원 평면을 쪼갠다
T(n) = 4 * T(n/4) + n^2
= O(n^2)

boards







Exercises

- *Matrix Exponentiation* using *Divide-and-Conquer*
  - Given an $n \times n$ square matrix A and a positive integer k, compute the power of the matrix A^k using a divide and conquer approach.
    - Matrix multiplication follows the following rule:
      - $c_{ij} = \left(\sum_{k=1}^{n} a_{ik}\, b_{kj}\right) \% \, 1{,}000$ for $1 \le i, j \le n$.
    - Use the following recursive relations:
      - $A^k = \begin{cases} I, if\ k = 0 \\ A, if\ k = 1 \\ A^{\frac{k}{2}} \times A^{\frac{k}{2}}, if\ k\ is\ even \\ A \times \left(A^{\left\lfloor \frac{k}{2} \right\rfloor} \times A^{\left\lfloor \frac{k}{2} \right\rfloor}\right), if\ k\ is\ odd. \end{cases}$

Any Questions?