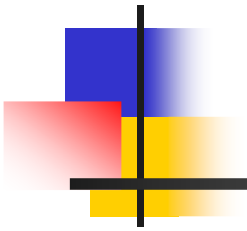
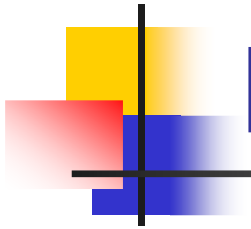


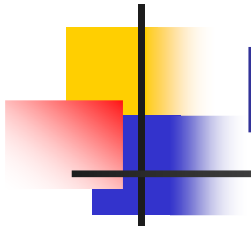
Ch2. Performance Analysis

SNU
IDB Lab.



Preview of Chapters

- Chapter 2
  - How to analyze the space and time complexities of program
- Chapter 3
  - Review asymptotic notations such as O, Ω, Θ, o for simplifying the analysis
- Chapter 4
  - Show how to measure the actual run time of a program by using a clocking method

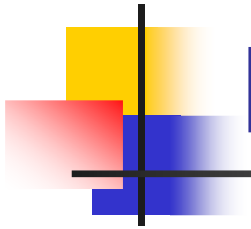


Bird's eye review (1/2)

- Correctness

  Even though the correctness of a program is most important,
  if the program takes unaffordable amount of memory and time,
  it is useless!

- Performance
  - Memory and time requirements



Bird's eye review (2/2)

- In this chapter
  - Paper and pencil methods to determine & analyze the memory and time requirements
  - Operation count & Step count
  - Best case, Worst case and Average case run time
  - Many application codes
    - Searching, Sorting, Matrix operations

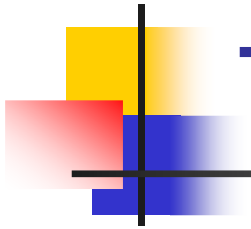
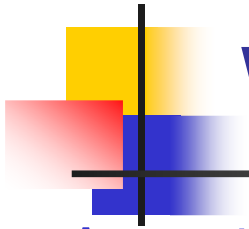


Table of Contents



What is complexity?

- Amount of computer memory and time needed to run a program
- Reasons for analyzing complexity: Multi-user computer system
  - **Space complexity**
    - Checking the memory available
    - Estimating the size of the largest problem that a program can solve
  - **Time complexity**
    - **The system requires the user to provide the upper limit on the program running time**
    - **If unaffordable, abort !**
    - **Need of real-time responses**
- Need to have the weighted measure of the time and space complexities of the alternative solutions

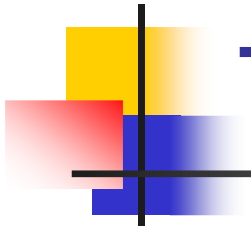
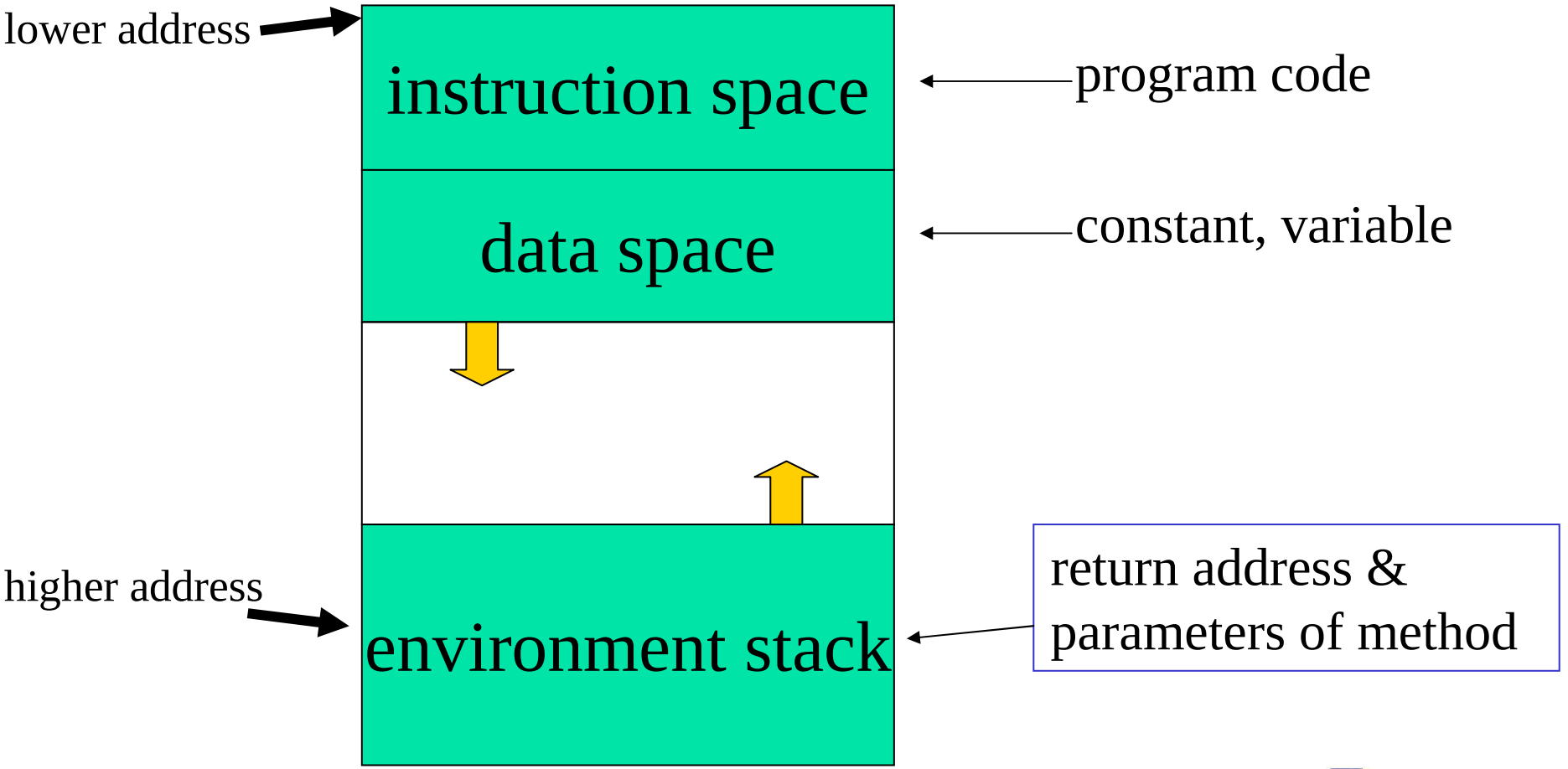
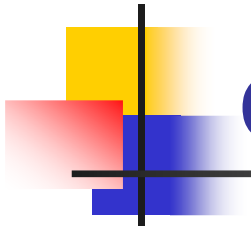


Table of Contents

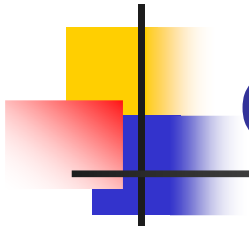
Memory structure of a program





Components of Space complexity (1/3)

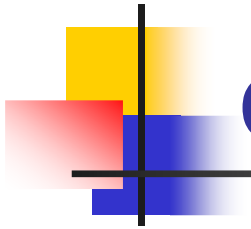
- Instruction space
  - Space needed to store the compiled version of the program instructions
  - Depending factors
    - Compiler
    - Compiler options
    - Target computer
- Data space
  - Space needed by constants and simple variables
  - Space needed by dynamically allocated objects
- Environment Stack (run time stack)
  - The return address
  - The values of all local variables and formal parameters in the method being involved



Components of Space complexity (2/3)

- Environment stack
  - General function calls: main → f → g → h
  - Recursive function calls: main → f → f …. →f
    - Factorial function: Main() → Fact(5) →Fact(4) … → Fact(1)

- Activation records
  - Arg1, Arg2, …, Argn
  - Caller's return address
  - Result Value
  - Local Variables
  - Saved registers



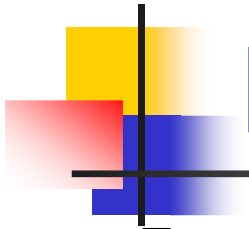
Components of Space complexity (3/3)

- The size of the instruction space is relatively insensitive to the particular problem instance being solved
- The data space needed by some of the dynamically allocated memory may also be independent of the problem size
- Our primary concerns
  - The recursion stack space and some of the dynamically allocated memory that depends on the instance characteristics and the problem size!
  - Yes! Run-Time Space Complexity

Sequential Search

```
public static int sequentialSearch (Object [] a, Object x) {
      int i;
      for ( i = 0; i < a.length && !x.equals(a[i]) ; i++ );
                  if (i == a.length) return -1;
                  else                     return i;
   }
```

- Space for array "a" is allocated where the actual parameter corresponding to "a" is declared and initialized
- Parameter is passing only reference
  - "Object[] a" passes the reference of a[0]
  - "Object x" passes the reference of x
- Space complexity of sequentialSearch is independent of a.length
  - Space complexity sequentialSearch (a.length) = 0
  - Even if "a" is long, there is no special space requirement in sequentialSearch



Factorial

- Factorial

```
public static int factorial(int n)
{
   if (n  <= 1)    return 1;
   else            return n * factorial(n-1);
}
```

  - Recursion depth will be max(n, 1)
  - Each time factorial is invoked, 8 bytes
    - Return address (4 bytes) + value of n (4 bytes)
  - Space complexity factorial (n) ➔ 8 * max(n,1)

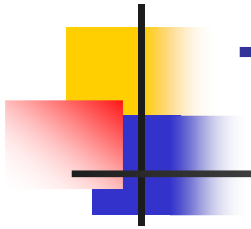
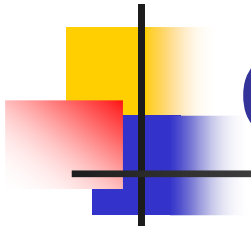


Table of Contents



Components of Time Complexity

- Compile time + Run time
  - Our primary concern is mainly run time
- One way to find out the accurate run time is to sum the operation times of machine instructions (ADD, SUB, DIV, MUL, …)
  - However, it is almost impossible
    - Data type (real, integer…)
    - Pipelining (hardware property)
- Instead we can estimate the run time using the followings
    - **Operation counts**
    - **Steps counts**

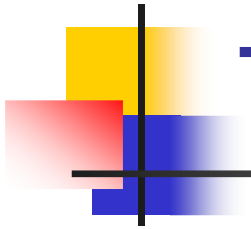
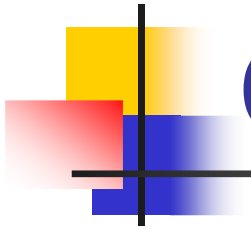


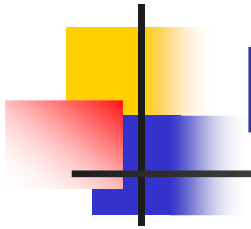
Table of Contents

- What is Performance?
- Space complexity
- Time complexity
  - Operations Counts
  - Step Counts



Operation Counts

- One way to estimate the time complexity
  - Select one or more operations such as add, multiply and compare
  - Determine how many of each is done
  - Identify the operations that contribute most to the time complexity

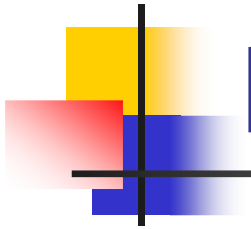


Max element

- Returns the position of the largest element in the array

```
public static int max(Comparable [] a, int n) {
   if (n < 0)    throw new IllegalArgumentException
                                   ("MyMath.max:Cannot find max of zero elements");
   int positionOfCurrentMax = 0;
   for (int i = 1; i <= n; i++)
        if ( a[positionOfCurrentMax].compareTo(a[i]) < 0 )
                positionOfCurrentMax = i;
        return positionOfCurrentMax;
}
```

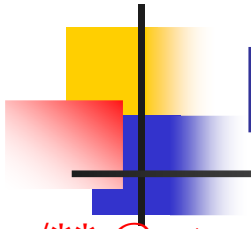
- Operation count → the number of comparisons between elements of the array "a"
  - When n <= 0, the for loop is not entered
  - When n > 0, each iteration of the for loop makes one comparison
- Operation count: Total number of comparisons is n
- Other operations are ignored (constant factor!)
  - Initializing positionOfCurrentMax
  - Incrementing for loop index i



Polynomial Evaluation

```
/** @return  coeff[0] * x^0 + coeff a[1] * x^1 + coeff[2] * x^2 + ...
    * @throws IllegalArgumentException  when coeff.length < 1 */
public static Computable valueOf(Computable [] coeff, Computable x)
{ if (coeff.length < 1)
  throw new IllegalArgumentException ("must have >= 1 coefficient");
  Computable y = (Computable) coeff[0].identity(),   // x^0
  value = coeff[0];                                  // coeff[0] * x^0
   // add remaining terms to value
      for (int i = 1; i < coeff.length; i++){
        y = (Computable) y.multiply(x);          // y = x^i
        value.increment(y.multiply(coeff[i])); // add in next term
      }
     return value;
   }
```

- **n = coeff.length - 1**
- **The number of additions is n**
- **The number of multiplications is 2n**



Horner's rule

```
/** @return coeff[0] * x^0 + coeff[1] * x^1 + coeff[2] * x^2 + ...
   * @throws IllegalArgumentException  when coeff.length < 1 */
  public static Computable valueOf(Computable [] coeff, Computable x)
  {
    if (coeff.length < 1)
       throw new IllegalArgumentException  ("must have >= 1 coefficient");
    // compute value
    Computable value = coeff[coeff.length - 1];
    for (int i = coeff.length - 2;  i >= 0;  i--){
      value = (Computable) value.multiply(x);
      value = (Computable) value.increment(coeff[i]);
    }
    return value;
  }
```

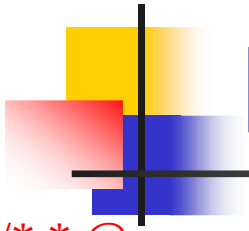
- **n = coeff.length - 1**
- **The number of additions is n**
- **The number of multiplications is n**

Horner's rule (설명)

- valueOf(): 다항식과 x의 값을 받아서 다항식의 값을 리턴
  - 예: coeff={1,2,3}이고 x=2이면, 1+2x+3x*x = 17을 리턴

- 3x*x+2x+1 ➔ ((3)x+2)x+1 이므로
- 최고차항부터 상수항까지 차수를 내리면서 아래처럼 계산

- 처음: value = 3 // 최고차항의 coefficient
- i=1: value = (3)*x+2 = 8
- i=0: value = (8)*x+1 = 17

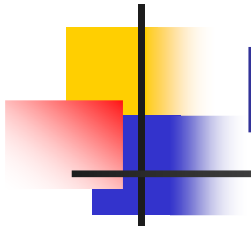


Ranking

```
/* * @param a is the array of objects to be ranked  * @param r is the array of computed ranks
    * @throws IllegalArgumentException  when the length of r is smaller than that of a */
public static void rank(Comparable [] a, int [] r)
{// Rank the objects in a[].
  // make sure rank array is large enough
  if (r.length < a.length)  throw new IllegalArgumentException
                                    ("length of rank array cannot " + "be less than the no of objects");
  for (int i = 0; i < a.length; i++)       r[i] = 0; // set all ranks to zero
  for (int i = 1; i < a.length; i++)     // compare all pairs of objects
       for (int j = 0; j < i; j++)
         if (a[j].compareTo(a[i]) <= 0)   r[i]++;
         else                                    r[j]++;
  }
```

Total element comparisons is

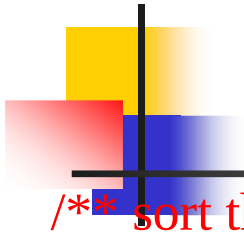
1+ 2 + 3 + … + n-1 = (n-1) * n / 2



Rank

- rank 함수는 리그전과 유사
  - 이중루프를 보면 모든 쌍 (i,j)에 대해 둘을 비교
  - a[j]가 a[i]보다 작으면 i의 승수가 증가 (r[i]++)
  - a[j]가 a[i]보다 크면 j의 승수가 증가.(r[j]++)

- 최종결과: r[i] = (i의 승수)
  - 원소들 중 a[i]보다 작거나 같은 원소의 갯수



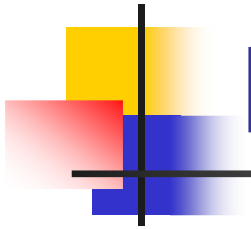
Rank Sort

```
/** sort the array a using the rank sort method */
public static void rankSort(Comparable [] a) {
int [] r = new int [a.length]; // create rank array
Rank.rank(a, r); // rank the elements
rearrange(a, r); // rearrange into sorted order
}
/** rearrange objects by rank using an additional array
    * @param a is the object array   @param r is the rank array */
 private static void rearrange(Comparable [] a, int [] r)
 { Comparable [] u = new Comparable [a.length]; // create an additional array u
      for (int i = 0; i < a.length; i++) u[r[i]] = a[i];   // move references to correct place in u
      for (int i = 0; i < a.length; i++) a[i] = u[i];      // move back to a
}
```

The complete sort requires

(n-1)*n / 2 comparisons

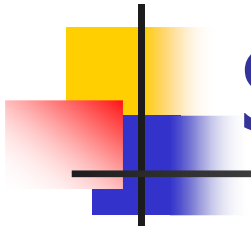
2n element reference moves



Rank Sort

- Rank Sort는 Rank()를 사용하는 Sort

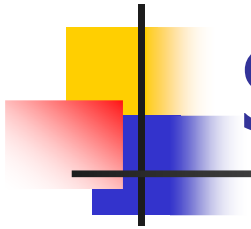
- a = {2,6,8,4} ➔ rank r={0,2,3,1}
- 첫 번째 for문
  - u[0]=2, u[2]=6, u[3]=8, u[1]=4 즉 u={2,4,6,8} 정렬
- 두 번째 for문
  - 단순히 u를 a에 복사



Selection Sort

```
/** sort the array a using the selection sort method */
  public static void selectionSort(Comparable [] a)
  {
    for (int size = a.length; size > 1; size--)
    {
      // find max object in a[0:size-1]
      int j = MyMath.max(a, size-1);

      // move max object to right end
      MyMath.swap(a, j, size - 1);
    }
  }
```

- Total number of comparisons (n-1) * n / 2
- Element reference moves 3*(n-1)



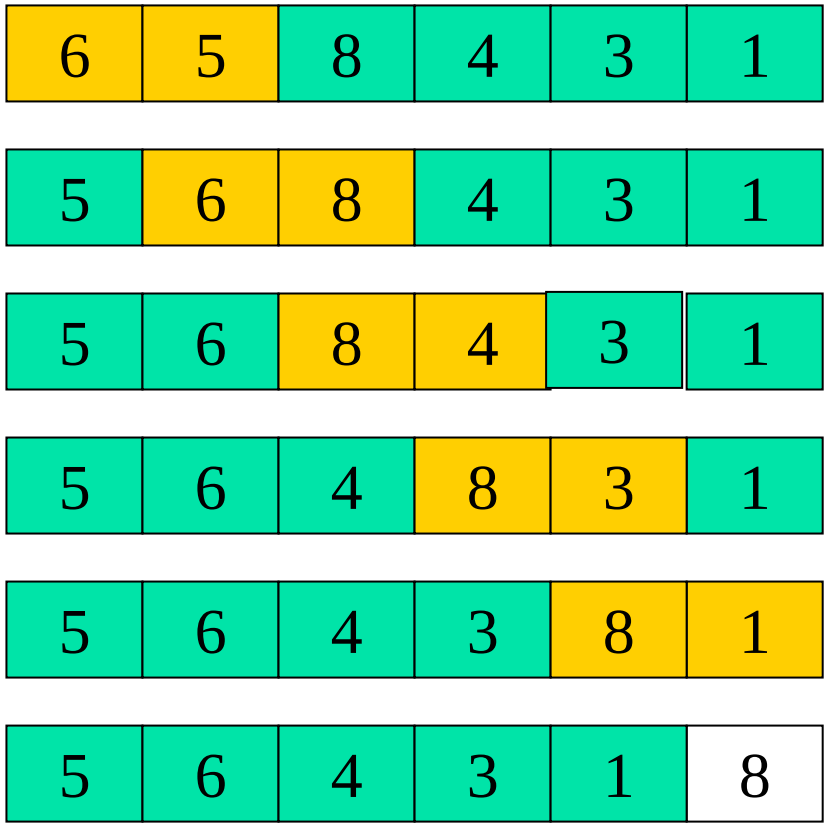
Selection Sort

- Idea: 나열된 숫자 중에서 가장 작은 숫자를 찾아서 맨 앞의 숫자와 교환
  - 처음에는 전체에서 가장 작은 숫자를 찾아서 첫번째 자리에 배치
  - 그 다음 두번째 숫자부터 끝까지 가장 작은 숫자를 찾아서 두번째 자리에
  - .....

예) 4 1 2 3 의 경우

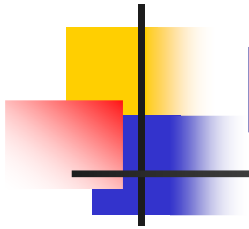
- Step 1 결과: 1 4 2 3 - [4 1 2 3] 에서 1이 가장 작으므로
- Step 2 결과: 1 2 4 3 - [4 2 3] 에서 2가 가장 작으므로
- Step 3 결과: 1 2 3 4 - [4 3] 에서 3이 가장 작으므로

Bubble Sort (1)



A bubbling pass

- A bubble sort consists of n-1 bubbling passes
- In a bubbling pass, pairs of adjacent elements are compared
- The elements are swapped to get the larger element to the right
- At the end of the bubbling pass, largest element in the right-most position

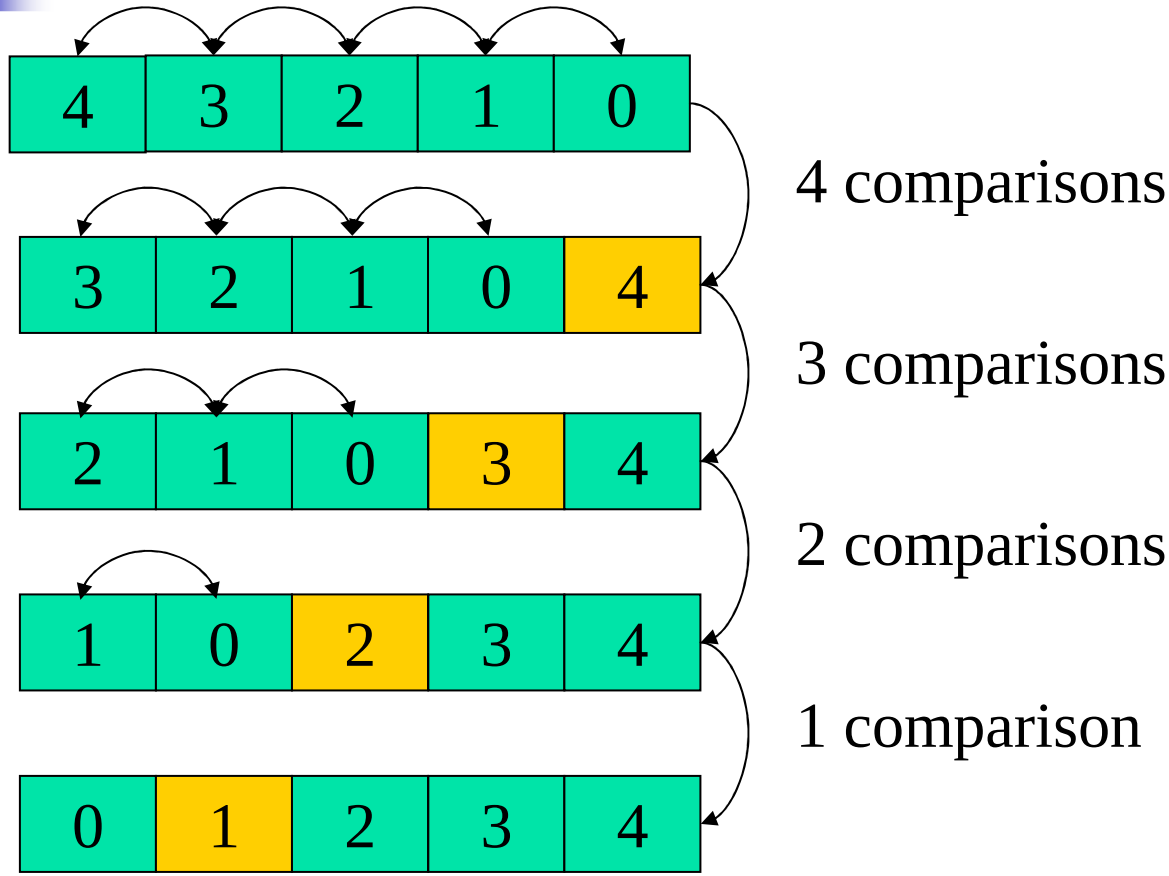


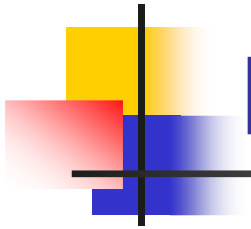
Bubble Sort (2)

```
private static void bubble(Comparable [] a, int n)
{
   for ( int i = 0; i < n-1; i++)
     if ( a[i].compareTo(a[n+1]) > 0)
                        MyMath.swap(a, i, i+1)
} // one bubbling pass


public static void bubbleSort(Comparable [] a)
{
   for ( int i = a.length; i > 1; i--)
                        bubble(a, i);
} // bubble sort
```

Maximum n(n-1)/2 comparisons

Bubble Sort example

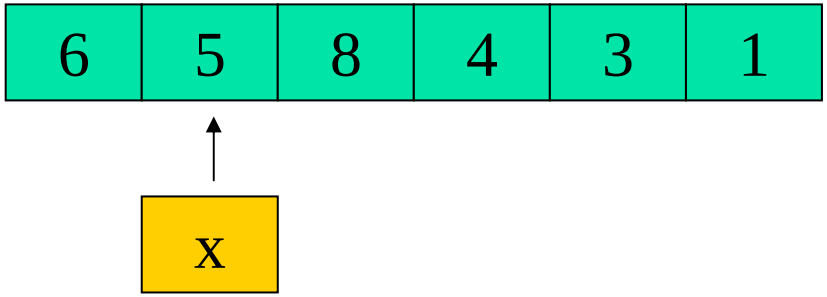




Best, Worst, and Average counts

- Operation counts
  - Not uniquely determined by the chosen instance characteristics (given data)
  - So, we generally ask for the best, worst, and average operation counts
  - Average operation count is often quite difficult to determine and define
  - We often limit our analysis to the best and worst operation counts

Sequential Search



Best case: when x is 6, comparison count *1*

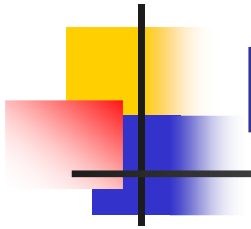
Worst case: when x is not in the array, comparison count *n*

Average case: $\frac{1}{n}\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Insertion into a Sorted Array

```
/** insert x into the sorted array a[0:n-1]     * a remains sorted after the insertion
   * @throws IllegalArgumentException when array a cannot accept x */
public static void insert(Comparable [] a, int n, Comparable x)
{
    if  (a.length < n + 1)
        throw new IllegalArgumentException   ("array not large enough");
   // find proper place for x
   int i;
   for (i = n - 1; i >= 0 && x.compareTo(a[i]) < 0; i--)      a[i+1] = a[i];
   a[i+1] = x;  // insert x
 }
```

- When x is inserted into position a[n-1], 1 comparison (best case)
- When x is inserted into position i+1, n-i comparisons
- When x is inserted into position a[0], n comparisons (worst case)



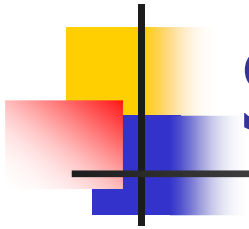
Rank Sort Revisited (1)

```
/** sort the array a using the rank sort method */
  public static void rankSort(Comparable [] a){
     // create rank array
     int [] r = new int [a.length];

     // rank the elements
     Rank.rank(a, r);

     // rearrange into sorted order
     rearrange(a, r);
  }
```

Rank Sort Revisited (2)

```
private static void rearrange(Comparable [] a, int [] r)
  {// In-place rearrangement into sorted order.
    for (int i = 0; i < a.length; i++)
      // get proper element reference to a[i]
      while (r[i] != i){
        int t = r[i];
        MyMath.swap(a, i, t);
        MyMath.swap(r, i, t);
      }
  }
```

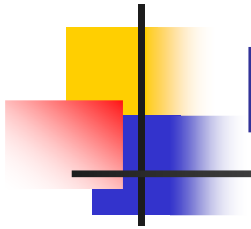
- Best case ➔ 0 swaps, Worst case ➔ 2(n-1) swaps
- Space requirements are smaller than (ex)2.10



Selection Sort Revisited

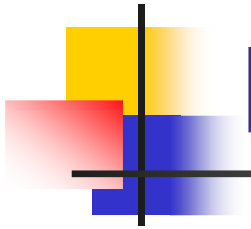
```
public static void selectionSort(Comparable [] a)
  {// Early-terminating version of selection sort.
    boolean sorted = false;
    for (int size = a.length; !sorted && (size > 1); size--)
    { int pos = 0;
      sorted = true; //when sorted no more iteration
     // find largest
      for (int i = 1; i < size; i++)
        if (a[pos].compareTo(a[i]) <= 0)  pos = i;
        else                                  sorted = false; // out of order
      MyMath.swap(a, pos, size - 1);
    }
  }
```

- Best case ➔ n-1 comparisons
- Worst case ➔ (n-1)*n / 2 comparisons



Bubble Sort Revisited (1)

```
// Early-terminating bubble sort
private static boolean bubble(Comparable [] a, int n) {

        boolean swapped = false; // no swaps so far
        for (int i = 0; i < n - 1; i++)

                if (a[i].compareTo(a[i+1]) > 0)
                {       MyMath.swap(a, i, i + 1);
                        swapped = true; // swap was done
                }

        return swapped;
}
public static void bubbleSort(Comparable [] a)
{       for (int i=a.length; i>1 && bubble(a,i); i--);
}
```



Bubble Sort Revisited (2)

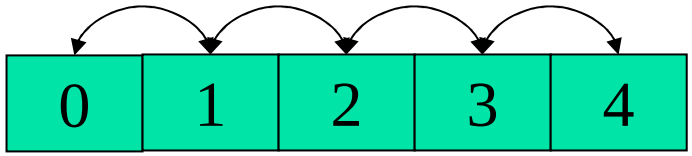
- If a bubbling pass results in no swaps, then the array is in sorted order and no further bubbling passes are necessary

- Best case
  - When sorting is already done
  - Number of comparisons ➔ n-1

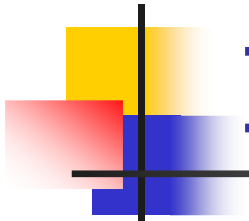
- Worst case
  - Same as original version
  - Number of comparisons ➔ n*(n-1) / 2

Bubble Sort example

- Best case
  - When sorting is already done



  - No swap occurs during 4 comparisons
  - Variable swapped remains false
  - Early terminating
- Worst case is as same as before
- Average case: difficult to measure



Insertion Sort

```
// insertion sort using Insert.insert in program 2.10
public static void insertionSort(Comparable [] a)
  {  for (int i = 1; i < a.length; i++)    Insert.insert(a, i, a[i]);  }

// another version of insertion sort
public static void insertionSort(Comparable [] a)
  { for (int i = 1; i < a.length; i++)
    {  Comparable t = a[i]; // insert a[i] into a[0:i-1]
       // find proper place for t
       int j;
      for (j = i - 1; j >= 0 && t.compareTo(a[j]) < 0; j--)    a[j+1] = a[j];
      a[j+1] = t;  // insert t = original a[i]
    }
  }
```

- Both versions of insertion sort
  - Best case ➔ n-1 comparisons
  - Worst case ➔ (n-1)*n / 2 comparisons

Insertion Sort

- Idea: 나열된 숫자들 중 두번째 자리의 수부터 차례대로 그 앞에 있는 숫자들과 비교해서 더 작을 경우 교환

- 예) 4 1 2 3 의 경우
- Step 1 : 두 번째 자리인 1을 앞에 있는 숫자들과 비교
  - 1 < 4 이므로 4와 교환 ➔ 1 4 2 3
- Step 2 : 세 번째 자리인 2를 앞에 있는 숫자들과 비교
  - 2 < 4 이므로 4와 교환 ➔ 1 2 4 3
  - 2 > 1 이므로 교환하지 않음
- Step 3: 네 번째 자리인 3을 앞에 있는 숫자들과 비교
  - 3 < 4 이므로 4와 교환 ➔ 1 2 3 4
  - 3 < 2 이므로 교환하지 않음. 그 앞의 수는 이전 과정에 의해 2보다 더 작은 수일 수밖에 없으므로 더 이상 비교하지는 않음.
-

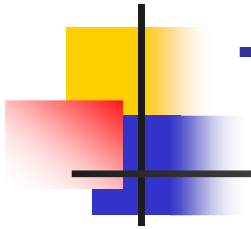
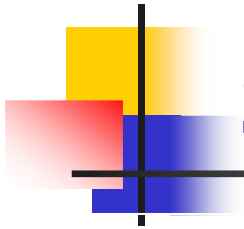


Table of Contents



Step counts for all prog statements

- Program Step
  - Definition 2.1: Syntactically or semantically meaningful segment of a program for which the execution time is independent of the input size
    - 10 additions or 100 divisions can be a step, but n additions cannot be
    - Return a+b*C+(a+b-c)/(a+b) + 4; // one step
    - x = y; // one step
- Step count method
  - Can account for the time spent in all parts of the program/method
    - More general than the operation count method
  - Can represent as functions of the instance characteristics
- Roughly, one meaning line of program is one step!

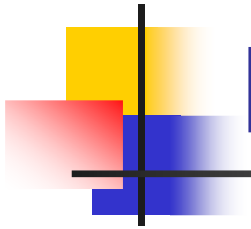
Counting steps in *sum()*

```
public static Computable sum (Computable [] a, int n)
{
    count++; // for conditional and return
    if (a.length == 0)   return null;
    Computable sum = (Computable) a[0].zero();
    count++; // for preceding statement
    for (int i = 0; i < n; i++)
      {   count++; // for the for statement
          sum.increment(a[i]);
          count++; // for increment
      }
    count++; // for last execution of for statement
    count++; // for return
    return sum;
 }
```

Each invocation of sum executes a total of 2n+4 steps when 0 <= n < a.length

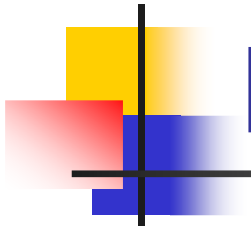
Counting steps for *rSum()*

```
public static Computable recursiveSum(Computable [] a, int n)
{// Driver for true recursive method rsum.
     count++; // for if-else statement
     if (a.length > 0) return rSum(a, n);
     else                  return null;  // no elements to sum
}
private static Computable rSum(Computable [] a, int n)
{ if (n == 0)
     {  count++; // for conditional and return
        return (Computable) a[0].zero();  }
   else
     {  count++;  // for conditional, rSum invocation, add, and return
       return (Computable) rSum(a, n - 1).add(a[n-1]); }
}
** The step count of recursiveSum is n+2 when 0 <=  n  < a.length
```



Profiling

- Rather than inserting "count" statement, list the total number of steps that each statement contributes to "count"
- s/e (steps per execution)
  - Number of steps per execution of the statement
    - The amount by which count changes as a result of the execution of that statement
  - x = sum(a, m) // s/e = 1+ (steps in sum) ➔ 1 + (2n + 4) ➔ 2n + 5
- Frequency
  - Total number of times each statement is executed
- Total number of steps = s/e X Frequency

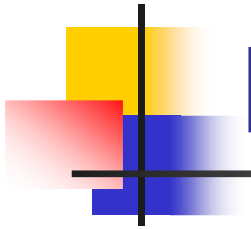


Matrix Transpose Profiling (1)

```
/** in-place transpose of matrix  a[0:rows-1][0:rows-1] */

public static void transpose(int [][] a, int rows)
  {
    for (int i = 0; i < rows; i++)
      for (int j = i+1; j < rows; j++)
      {
        // swap a[i][j] and a[j][i]
        int  t      = a[i][j];
             a[i][j] = a[j][i];
             a[j][i] = t;
      }
  }
```

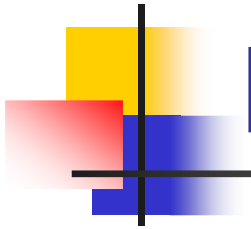
Matrix Transpose Profiling (2)

Statement	s/e	Frequency	Total steps
public static void transpose(…)	0	0	0
{	0	0	0
for (int i = 0; i < rows; i++)	1	rows+1	rows+1
for (int j = i+1; j < rows; j++)	1	rows(rows+1)/2	rows(rows+1)/2
{	0	0	0
// swap a[i][j] and a[j][i]	0	0	0
int t = a[i][j];	1	rows(rows-1)/2	rows(rows-1)/2
a[i][j] = a[j][i];	1	rows(rows-1)/2	rows(rows-1)/2
a[j][i] = t;	1	rows(rows-1)/2	rows(rows-1)/2
}	0	0	0
}	0	0	0
Total			$2\,rows^{2}+1$



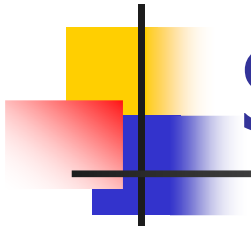
Prefix Sums profiling (1)

```
// Inefficient Prefix Sum
public static int [] inef(int [] a)
{
    // create an array for the prefix sums
    int [] b = new int [a.length];

    // compute the prefix sums
    for (int j = 0; j < a.length; j++)
        b[j] = MyMath.sum(a, j + 1);

    return b;
}
```



Prefix Sums profiling (2)

Statement	s/e	Frequency	Total steps
public static int [] inef(int [] a)	0	0	0
{	0	0	0
// create an array for the prefix sums	0	0	0
int [] b = new int [a.length];	n	1	n
	0	0	0
// compute the prefix sums	0	0	0
for (int j = 0; j < a.length; j++)	1	n+1	n+1
b[j] = MyMath.sum(a, j + 1);	2j+7	n	n(n+6)
return b;	1	1	1
}	0	0	0
Total			n^2+8n+2



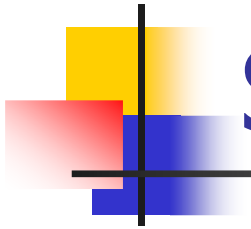
Sequential Search profiling (1)

```
public static int sequentialSearch (Object [] a, Object x) {

    int i;
    for (i = 0; i < a.length && !x.equals(a[i]); i++);

        if (i == a.length) return -1;
        else               return i;
}
```

Sequential Search profiling (2)

Statement	s/e	Frequency	Total steps
public static int sequentialSearch(…)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = 0; i < a.length&& !x.equals(a[i]); i++);	1	1	1
if (i == a.length) return -1;	1	1	1
else return i;	1	1	1
}	0	0	0
Total			4

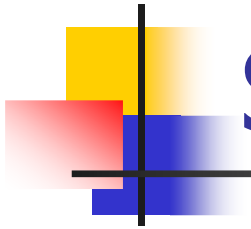
Best-case step count



Sequential Search profiling (3)

Statement	s/e	Frequency	Total steps
public static int sequentialSearch(…)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = 0; i < a.length&& !x.equals(a[i]); i++);	1	n+1	n+1
if (i == a.length) return -1;	1	1	1
else return i;	1	0	0
}	0	0	0
Total			n+3

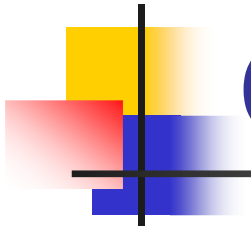
Worst-case step count



Sequential Search profiling (4)

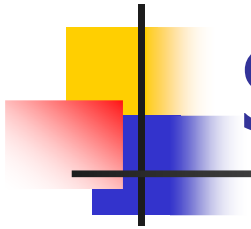
Statement	s/e	Frequency	Total steps
public static int sequentialSearch(…)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = 0; i < a.length&& !x.equals(a[i]); i++);	1	j+1	j+1
if (i == a.length) return -1;	1	1	1
else return i;	1	1	1
}	0	0	0
Total			j+4

Step count when x = a[j]



Observation!

- We want to to predict the growth of execution time in run time & compare the time complexities of two programs
- But…….
- Operation count might ignore some important parts of the program
- The notion of step is rather random
- We need a general, easy, and intuitive methodolody for measuring time complexity! ➔ asymptotic analysis



Summary

- Correctness
  - Even though the correctness of a program is most important,
  - if the program takes unaffordable amount of memory and time,
  - it is useless!
- Performance
  - Memory and time requirements
- In this chapter
  - Paper and pencil methods to determine & analyze the memory and time requirements
  - Operation count & step count
  - Best, worst and average case run time
  - Many application codes
    - Searching, Sorting, Matrix operations