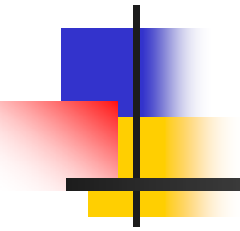
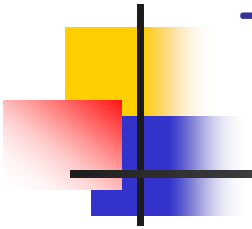


Basic Concepts

Introduction to Data Structures

Kyuseok Shim
SoEECS, SNU.



Topics

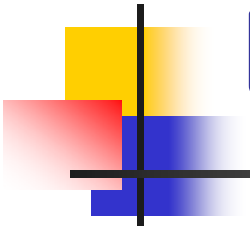
- General course information
  - TAs, course web site
  - grading, exams, assignments
  - class schedule
- Introduction to data structures
- Introduction to algorithm analysis

General information

- Instructor: 심 규 석
  - Office hour: by prior appointment thru e-mail
  - ☎: 880-7269, Email: shim@ee.snu.ac.kr
- TA: 김영훈(*head TA*) , 김영훈, 장지훈, 손승락, 김진현
  - ☎: 880-1758, Email: ds-ta@kdd.snu.ac.kr
- References
  - Lecture notes
  - E. Horowitz, et al, Fundamentals of data structures in C++ (2^{nd} ed.)
  - Reference books
    - M. Goodrich, et al, Data Structures and Algorithms in Java/C++

Grading policy

- Two exams: 69 %
  - midterm (29%)
  - final (40%)
- Assignments: 28%
  - about 3~4 assignments
  - mostly simple programming assignments using C++ (or Java)
    → We are expecting you already took Programming Methodologies or related courses, and learned *object-oriented programming*.
  - Details will be given by the TAs later.
- Class attendance: 3%
  - Attendance sheets will be handed out during the class.
  - Please sign it up for your attendance verification.
    → But, Never do it for your friend(s)!!

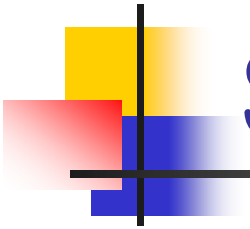


Lecture information

- Organization of the lecture
  - 70~80 min lecture with possible 10 min interim breaks
  - 1:00~2:15 pm, Mon. and Wed.
  - Regular lectures + Programming lab hours
- Expected grading distribution for this course – absolute grading will be given

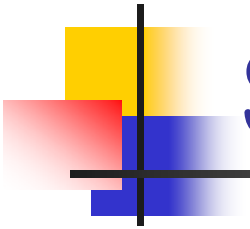
Tentative class schedule

구 분	강 의 내 용	과제 시험
1 주	Introduction	숙제1
2 주	Algorithm analysis	
3 주	Arrays & Linked-list	
4 주	Arrays & Linked-list	
5 주	Stacks	숙제2 ->중간고사
6 주	Queues	
7 주	Trees	
8 주	Trees	
9 주	Sets	
10 주	Heaps	
11 주	Graphs	숙제3
12 주	Graphs	
13 주	Strings	숙제4
14 주	Basic Algorithms: Hashing	
15 주	Basic Algorithms: Sorting	



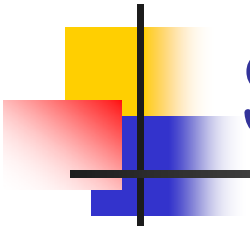
System Life Cycle

- Requirements
- Specification
- Design
- Coding
- Verification
- Operation and maintenance



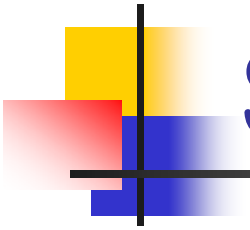
System Life Cycle

- Requirements
  - Gathering
    - Types of expression
  - Analysis
    - Elimination of ambiguities
    - Partition of requirements



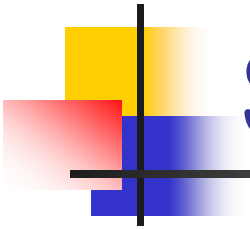
System Life Cycle

- Specification
  - Define what the system is to do
  - Functional specification
  - Performance specification
  - Use specification language



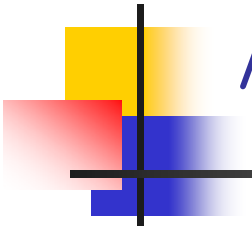
System Life Cycle

- Design
  - Description of how to achieve the specification
  - Data objects → Abstract data types
  - Operations → Algorithms
  - Programming language independent
  - Design approaches
    - Top-down approach
    - Bottom-up approach



System Life Cycle

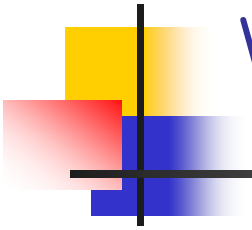
- Coding
  - Programming language dependent
- Verification
  - Correctness proofs
  - Test
  - Error removal
- Operation and maintenance
  - System installation and operation
  - System modification
  - Major item for cost



Algorithm Analysis

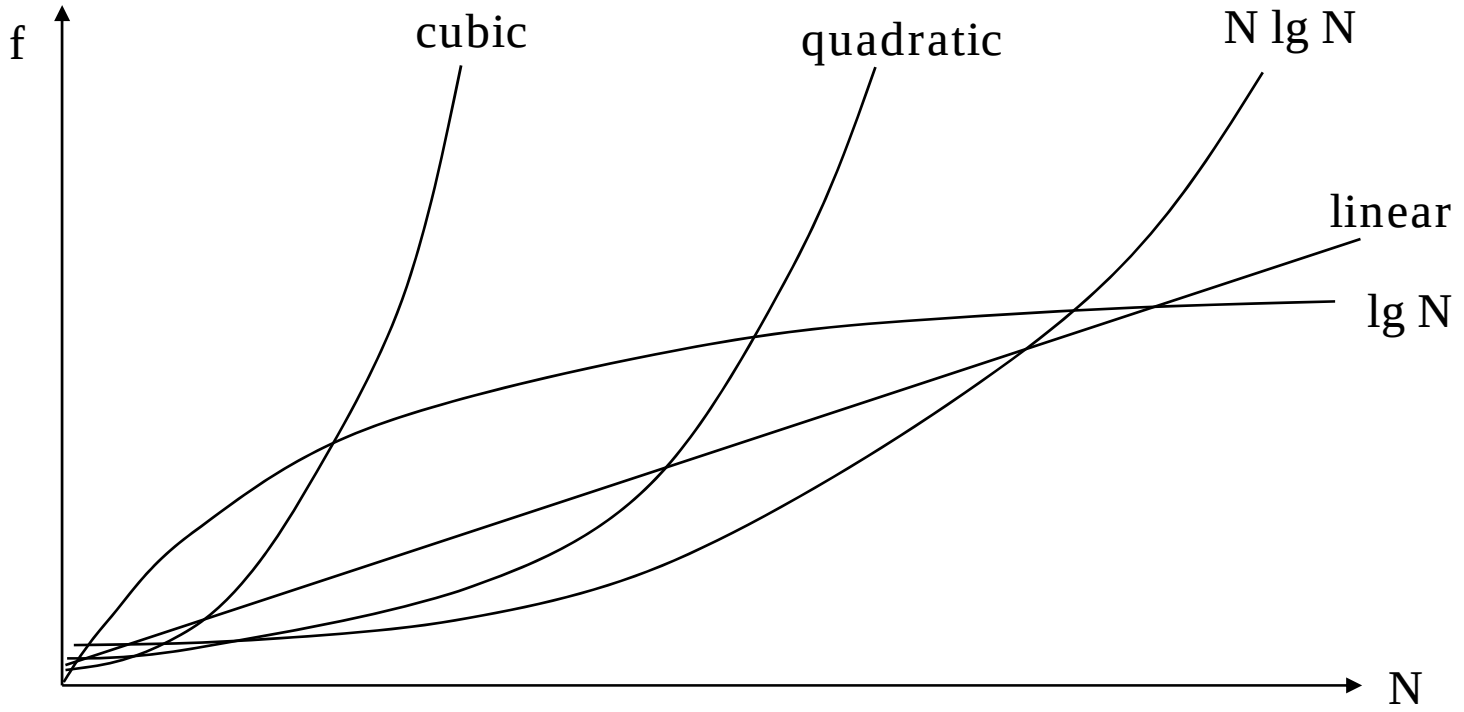
- Running time of an algorithm almost always depends on the amount of input: More input means more time. Thus the running time, T, is a function of the amount of input, N, or $T(N) = f(N)$.
- The exact value of the function depends on
  - the speed of the host machine;
  - the quality of the compiler and optimizer;
  - the quality of the program that implements the algorithm;
  - the basic fundamentals of the algorithm
- Typically, the last item is most important.

Worst-case Vs. Average Case

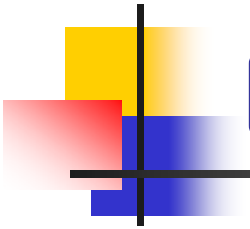


- Worst-case running time is a bound over all inputs of a certain size *N*. (Guarantee)
- Average-case running time is an average over all inputs of a certain size *N*. (Prediction): Difficult to define the distribution to compute the average cases
- Best case running time: Can be used to argue that the algorithm is really bad.

Running Time Functions

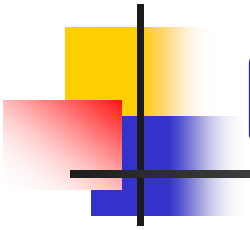


- Several common functions; for small inputs some are somewhat faster than others.



Definitions

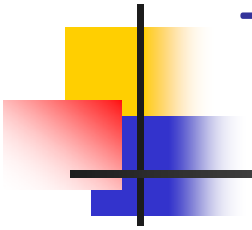
- Big-Oh for upper bound:
  - T(N) = O(f(N)) if there area positive constants c and n0 such that T(N) <= c f(N) when N >= n0.
- Big-Omega for lower bound:
  - T(N) = Omega(g(N)) if there are positive constants c and n0 such that T(N) >= c g(N) when N >= n0.
- Big-Theta:
  - T(N) = Theta(h(N)) iff T(N) = O(h(N)) and T(N) = Omega(h(N)).
- Small-Oh:
  - T(N) = o(p(N)) iff T(N) = O(p(N)) and T(N) is not Theta(p(N)).



Properties of Big-Oh

- If T1(N) = O(f(N)) and T2(N) = O(g(N)), then
  - T1(N) + T2(N) = max(O(f(N)), O(g(N))
    - Lower-order terms are ignored
  - T1(N)*T2(N) =O(f(N)*g(N))
- O(c f(N)) = O(f(N)) for some constant c
  - Constants are ignored!
- In reality, constants and lower-order terms may matter, especially when the input size is small.
- Can you prove the above properties with the definitions?

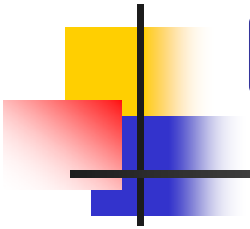
Types of Functions; Big-Oh



- Cubic: dominant term is some constant times N^3. We say $O(N^3)$.
- Quadratic: dominant term is some constant times N^2. We say $O(N^2)$.
- $O(N \log N)$: dominant term is some constant times $N \log N$.
- Linear: dominant term is some constant times N. We say $O(N)$.
- Example: $350N^2 + N + N^3$ is cubic.
- Big-Oh ignores leading constants.

Dominant Term Matters

- Suppose we estimate $350N^2 + N + N^3$ with N^3.
- For $N = 10000$:
  - Actual value is 1,003,500,010,000
  - Estimate is 1,000,000,000,000
  - Error in estimate is 0.35%, which is negligible.
- For large N, dominant term is usually indicative of algorithm's behavior.
- For small N, dominant term is not necessarily indicative of behavior, BUT, typically programs on small inputs run so fast we don't care anyway.



Basic Issues

- How much better is one curve than another (answer: typically a lot, for large inputs)
- How do we decide which curve a particular algorithm lies on (answer: sometimes it's easy, sometimes it's hard).
- Can we use this information to design better algorithms (answer: definitely).
- Can we predict how an algorithm will perform for large input sets, based on its performance for moderate input sets (answer: definitely).

Running Time Calculation

1. Summations for Loops

```
for  i = 1 to n do {                for  i = 1 to n do {
        . . . .                             for j = 1 to n do {
        . . . .                                  . . . . . .
```

If the loop of (a) takes $f(i)$ times, $T(n) = \sum_{i=1}^{n} f(i)$

If the loop of (b) takes $g(i, j)$ times, $T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} g(i, j)$

$\sum_{i=1}^{n} 1 = n$ **constant sum**

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ **the linear sum**

$\sum_{i=1}^{n} c^i = \frac{c^{n+1} - 1}{c - 1}$, $c \neq 1$

Running Time Calculation

2. Sequential and If-Then-Else Blocks

```
for i = 1 to n do {
    A[i] = 0;
}
for i = 1 to n do {
    for  j = 1 to n do {
      A[i]++;
    }
}
```

$$T(n) = O(n) + O(n^2) = O(n^2)$$

```
if (cond)
   S1
else
   S2
```

$$T(n) = max(T_{s1}(n),\ (T_{s2}(n))$$

Running Time Calculation

3. Recursion

```
unsigned long int
Factorial (const unsigned long N)
{
        if (N<=1)  return 1;
        else return n* Factorial(N-1);
}
```

$$T(n) = 1 \quad \text{when} \quad n \leq 1$$
$$T(n) = 1 + T(n-1) \quad \text{when} \quad n > 1$$

$$T(n) = 2 + T(n-2) = 3 + T(n-3) = \ . \ . \ . \ = (n-1) + T(1)$$

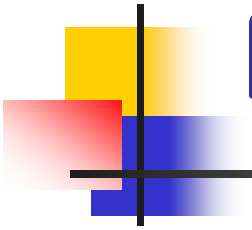
$$T(n) = O(n)$$

Analysis of the algorithm

$$T(1) = 1$$
$$T(n) = 2T(\tfrac{n}{2}) + n$$

Let $n = 2^k$. Then,

$$T(n) = 2T(\frac{n}{2}) + n = 2(2T(\frac{n}{2^2}) + \frac{n}{2}) + n =$$
$$= 2^2 T(\frac{n}{2^2}) + 2n = 2^2(2T(\frac{n}{2^3}) + \frac{n}{2^2}) + 2n =$$
$$= 2^3 T(\frac{n}{2^3}) + 3n = \ldots$$
$$= 2^k T(\frac{n}{2^k}) + kn = n + n \log n$$



Proof by Induction

The most Important Proof Technique in all aspects of Computer Science and Data Structures.

The Proof is composed of Two Steps

- Base step: prove the theorem is true for the initial value(s)
- Induction step: assume that the steorem holds for all values up to n, then show that the theorem holds for the next value n+1.

Proof by Induction–Example

Fibonacci Numbers:

$F0 = F1 = 1, Fi = Fi-1 + Fi-2 (i > 1)$

Prove that $Fi < (\frac{5}{3})^i$ for $i > 1$ (Weiss1.2. 5)

Base : $F1 = 1 < (\frac{5}{3})^1$ and $F2 = 2 < (\frac{5}{3})^2$

Induction : Assume that the theorem is true for $i = 1,2,...k$

show that $Fk+1 < (\frac{5}{3})^{k+1}$

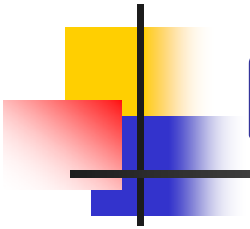
We know that $Fk+1 = Fk + Fk-1$

$Fk+1 < (\frac{5}{3})^k + (\frac{5}{3})^{k-1}$

$Fk+1 < \frac{3}{5}(\frac{5}{3})^{k+1} + \frac{9}{25}(\frac{5}{3})^{k+1}$

$Fk+1 < \frac{24}{25}(\frac{5}{3})^{k+1}$

$Fk+1 < (\frac{5}{3})^{k+1}$



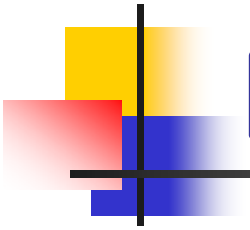
Recurrences

- Mathematical Induction
- Recursion-tree method
- Solve the recurrence relations
  - Method 1
  - Method 2

Mathematical Induction

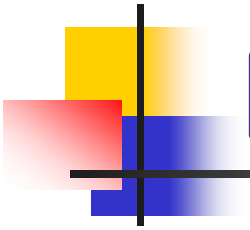
- Guess the form of the solution
- Use mathematical induction to find the constants and show that the solution works.
- E.g. $T(n) = 2T(n/2) + n$ and prove $T(n) <= c\ n \log n$
  - Basis: $T(1) = 1$
  - Assumption: $T(n) <= c\ n \log n$ for $n < k$
  - Induction step:

$$
\begin{aligned}
T(k) = 2T(k/2)+k &<= 2\ c\ (k/2) \log(k/2) + k \\
&= c\ k \log(k/2) + k \\
&= c\ k \log k - ck \log 2 + k \\
&= c\ k \log k - ck + k \\
&<= c\ k \log k
\end{aligned}
$$



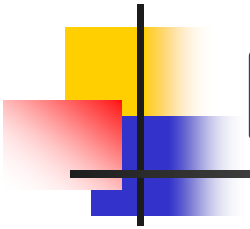
Mathematical Induction

- Basis case revisited
  - T(1) <= c 1 log 1 = 0 wrong!!
  - What should we do?
    - We are interested in asymptotic behavior
    - Remove the difficult boundary condition from induction proof
    - T(n) <= c n log n for n >= 2
      - T(1) = 1
      - T(2) = 2T(1) + 2 = 4 <= c 2 log 2 = 2c for c >= 2



Mathematical Induction

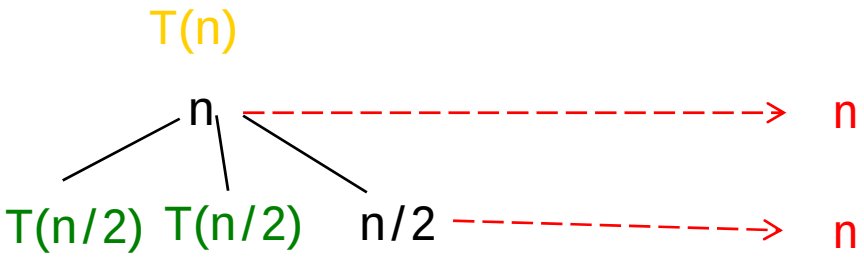
- T(n) = T(n/2) + T(n/2) + 1, show T(n) <= c n
  - Basis: T(1) = 1 <= c
  - Assumption: T(n) <= c n for n < k
  - Induction step:
    - T(n) = 2 T(n/2) + 1 <= c n + 1
    - Does not imply T(n) <= c n
    - Use a new guess: T(n) <= c n – b
    - T(n) <= c n – 2 b + 1 <= c n – b as long as b >= 1



Mathematical Induction

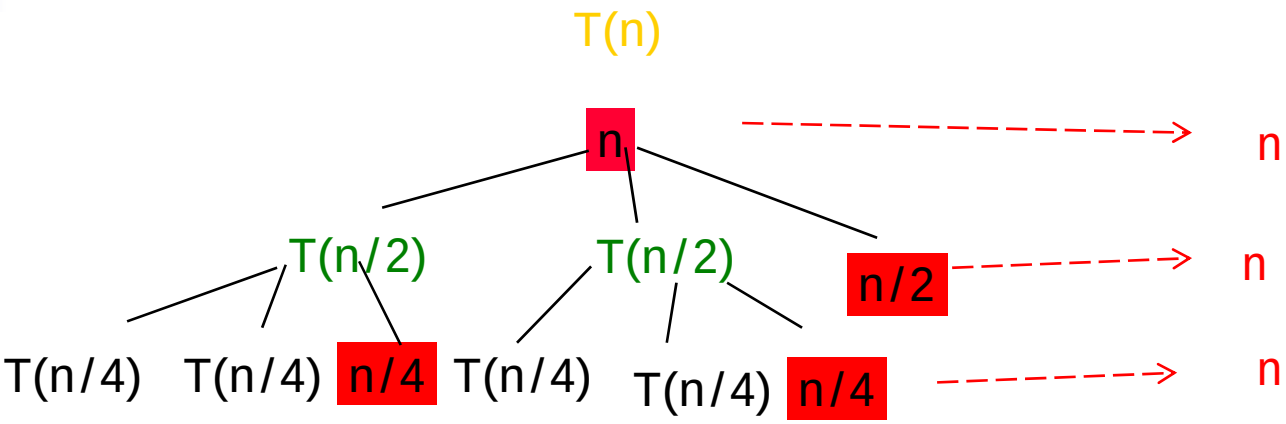
- Avoiding Pitfalls
  - $T(n) = 2T(n/2) + n$ and prove $T(n) <= c\ n$
  - $T(k) <= 2\ c\ (k/2) + k = c\ k + k$
    - Thus, $T(n) <= c\ n$.
    - Wrong!! We should prove the exact form of the induction hypothesis, that is $T(k) <= c\ k$.
- Changing variables
  - $T(n) = 2\ T(n^{1/2}) + \log n$
  - Let $m = \log n$, $T(2^m) = 2\ T(2^{m/2}) + m$
  - Then, let $S(m) = T(2^m)$, $S(m) = 2\ S(m/2) + m$
  - $S(m) = O(m \log m)$
  - $T(n) = T(2^m) = S(m) = O\ (m \log m) = O(\log n \log (\log n))$.

Recursion-tree method



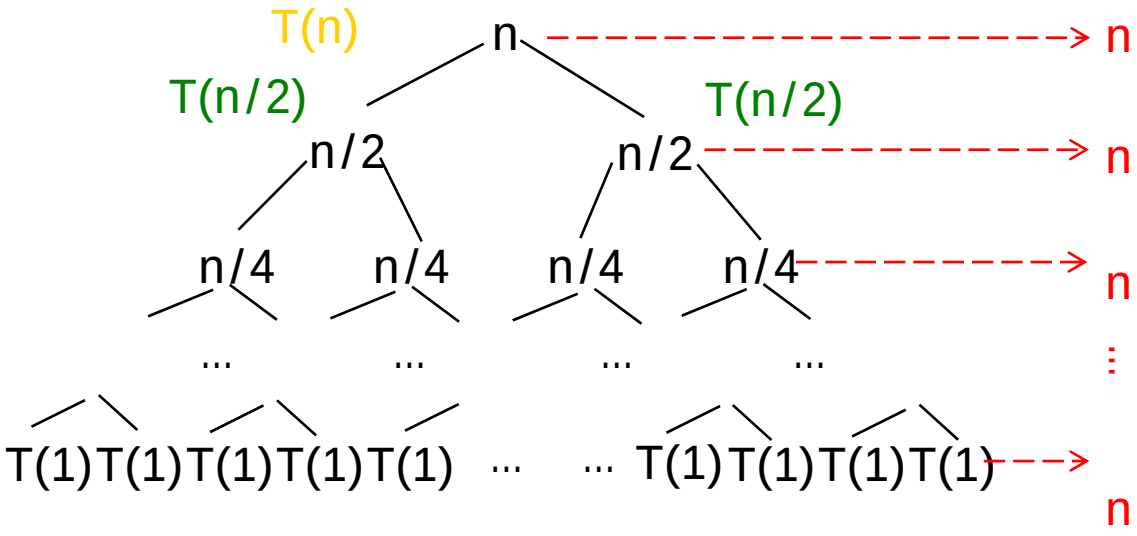
Ex: $T(n)=2T(n/2)+n$

Recursion-tree method



Ex: T(n)=2T(n/2)+n

Recursion-tree method



Ex: $T(n)=2T(n/2)+n$

$= n+n+\ldots+n$

$= O(n \lg n)$

Solve recurrence relation (1)

- $T(1)$, $T(n) = 2T(n/2)+n$

$T(n) = 2T(n/2)+n$

$= 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n$

$= \ldots\ldots$

$= 2^k T(n/(2^k)) + k\, n$

Let $k = \log n$

We have $T(n) = n\, T(1) + n \log n = n + n \log n$

Solve recurrence relation (2)

- T(1), T(n) = 2T(n/2)+n

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

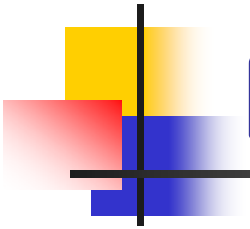
$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

$$\Rightarrow \quad \frac{T(N)}{N} = \frac{T(1)}{1} + \log N$$

.....

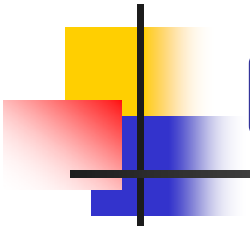
Thus, $T(N) = N \log N + N = O(N \log N)$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$



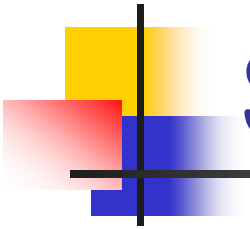
Recursive Algorithms

- Direct recursion : functions call themselves
- Indirect recursion : functions call other functions that invoke the calling function
- Recursive mechanisms can express an otherwise complex process very clearly
- Appropriate cases
  - when the problem itself is recursively defined
  - when the data structure for the algorithm is recursively defined



Description of an Algorithm

- A natural language like English
  - peudo code
- Programming language
  - C++



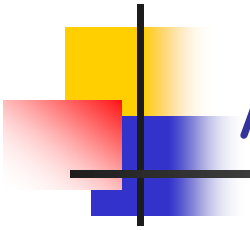
Selection Sort

- A program that sorts a collection of $n \geq 1$ integers
- From those integers that are currently unsorted, find the smallest and place it next in the sorted list

Selection Sort

```
for (int i = 0; i < n; i++) {
   examine a[i] to a[n-1] and suppose the smallest integer  is at a[j];
   interchange a[i] and a[j];
}
```

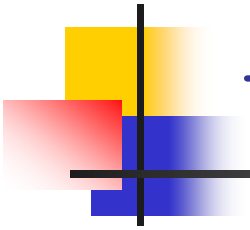
```
1   void sort(int *a, const int n)
2   // sort the n integers a[0] to a[n-1] into nondecreasing  order
3   {
4         for (int i = 0; i < n; i++)
5         {
6               int j = i;
7               // find smallest integer in a[i] to a[n-1]
8               for (int k = i + 1; k < n; k++)
9                  if (a[k] < a[j]) j = k;
10              // interchange
11              int temp = a[i]; a[i] = a[j]; a[j] = temp;
12        }
13  }
```



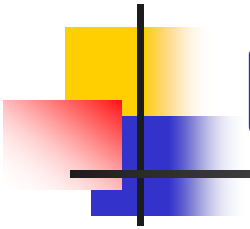
An Example of Steps

	a[0]	a[1]	a[2]	a[3]
	5	4	13	2
i=0	2	4	13	5
i=1	2	4	13	5
i=2	2	4	5	13

Correctness Proof Method for Loops

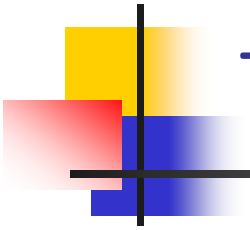


- Develop Loop Invariant
- Steps of proof – show the following
  - Initialization: It is true prior to the first iteration of the loop
  - Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration
  - Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct



Loop Invariant

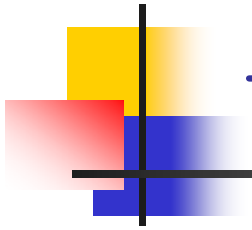
- You run a loop in order to change things
- Oddly enough, what is usually most important in understanding a loop is finding an invariant: that is, *a condition that doesn't change*



Theorem

- sort(a, n) correctly sorts a set of n≥1 integers

Correctness Proof Method for Loops (Wrong!!!)

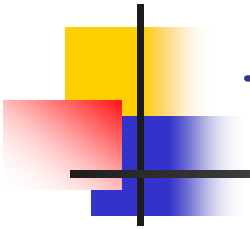


- Loop Invariant
  - At the start of each iteration for the for-loop of lines 3-10, the subarray a[0..i-1] are sorted such that a[0] <= a[1] <= … <= a[i-1]
- Steps of proof – show the following
  - Initialization: It is true prior to the first iteration of the loop
  - Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration
  - Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

Correctness Proof Method for Loops

- Loop Invariant
  - At the start of each iteration for the for-loop of lines 3-10, the subarray a[0..i-1] are sorted such that
    - (1) a[0] <= a[1] <= ··· <= a[i-1]
    - (2)' a[i-1] <= a[m] for all m such that i <= m <= n-1
    - (3) all values in a[i] with 0 <= i <= n-1 are from the initial array a[0..n-1] and ****
- Steps of proof – show the following
  - Initialization:
  - Maintenance:
  - Termination:

Correctness Proof Method for Loops

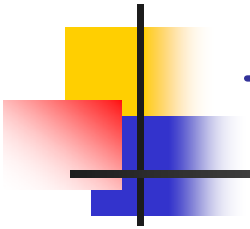


- Loop Invariant
  - After i iterations,
    - (1) the array still contains all the same elements as before, and
    - (2) the array positions a[j] for $0 \leq j \leq i-1$ contain the j-th smallest elements, sorted.
- Steps of proof – show the following
  - Initialization:
  - Maintenance:
  - Termination:

Correctness Proof Method for Loops

- Initialization:
  - With i = 0, the algorithm has not done anything yet, so the array, of course, is still unchanged, and the first 0 elements contain what they should (since there are no elements)

Correctness Proof Method for Loops



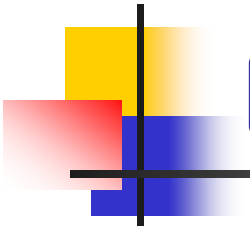
- Maintenance:
  - We assume that the invariant held for i (i.e before start of the loop), and need to prove that after one more iteration (before next iteration), it holds for i + 1.
  - For part (1), by induction hypothesis, the array still contained the same numbers after i iterations. The only way in which the array is changed is by the exchange operations, which only changes the order of elements in an array, but does not overwrite anything. So the array still contains the same elements as before after i + 1 iterations.

Correctness Proof Method for Loops

- Maintenance:
  - We assume that the invariant held for i, and need to prove that after one more iteration, it holds for i + 1.
  - For part (2), we need to show that the first i+1 array positions contain the i+1 smallest elements, in the correct order.
  - For the first i elements, we can argue that the array already contained the correct elements after i iterations (by Induction Hypothesis for i), and iteration number i+1 does not access or overwrite or swap those positions, so they still contain the right elements.
  - To prove that the element in position i+1 is the right one, we notice that the inner loop finds the smallest element between positions i , . . . , n (this could be proved by another induction, on j).
  - Because the i smallest elements are in positions 0, . . . , i−1, the smallest among the remaining between positions i , . . . , n is the (i + 1)−smallest element overall, which is then swapped into position i, so that that position contains the right element. This completes the inductive proof.

Correctness Proof Method for Loops

- Termination:
  - Loop Invariant
    - After i iterations,
      - (1) the array still contains all the same elements as before, and
      - (2) the array positions a[j] for $0 \le j \le i-1$ contain the (j+1)−th smallest elements, sorted.
  - Since i = n now, we have
      - (1) the array still contains all the same elements as before, and
      - (2) the array positions a[j] for $0 \le j \le n-1$ contain the (j+1)−th smallest elements, sorted.
  - Thus, we proved



Binary Search

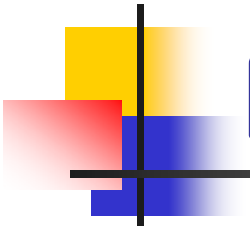
- n≥1 distinct integers are already sorted in the array a[0]...a[n-1]
- If the integer x is present, return j such that x=a[j] otherwise return
- left, right : the left and right ends of the list to be searched(initially:0, n-1)
- middle = (left + right)/2
- compare a[middle] with x
  - x<a[middle] : right = middle-1
  - x=a[middle] : return middle
  - x>a[middle] : left = middle+1
- two subtasks
  - determine if there are any integers left to check
  - compare x to a[middle]

C++ Function for Binary Search

```
char compare(int x, int y)
{
        if (x > y) return '>';
        else if (x < y) return '<';
                else return '=';
        } // end of compare
}
 1   int BinarySearch(int *a, const int x, const int n)
 2   // Search the sorted array a[0], ..., a[n-1] for x
 3   {
 4           for (int left = 0, right = n-1; left <= right;) { // while more elements
 5                   int middle = (left + right) / 2;
 6                  switch (comare(x,a[middle])) {
 7                          case '>' : left = middle + 1; break; // x > a[middle]
 8                          case '<' : right = middle - 1; break; // x < a[middle]
 9                          case '=' : return middle; // x == a[middle]
10                   } // end of switch
11           } // end of for
12           return -1; // not found
13   } // end of BinarySearch
```

Recursive Binary Search

```
int BinarySearch(int *a, const int x, const int left, const int right)
// Search the sorted array a[left], ..., a[right] for x
{
        if(left <= right) {
                int middle = (left + right) / 2;
                switch(compare(x, a[middle])) {
                        case '>' : return BinarySearch(a, x, middle+1, right);
       // x > a[middle]
                        case '<' : return BinarySearch(a, x, left, middle-1);
       // x < a[middle]
                        case '=' : return middle; // x == a[middle]
                } // end of switch
        } // end of if
        return -1; // not found
} // end of BinarySearch
```



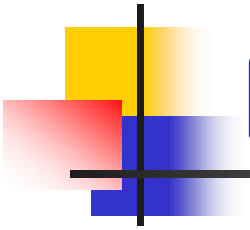
Performance Analysis

- Program Complexity
  - Space complexity – the amount of memory it needs
  - Time complexity – the amount of computer time it needs
- Performance evaluation phases
  - performance analysis – a priori estimates
  - performance measurement – a posteriori testing

Iterative vs. Recursive function for sum

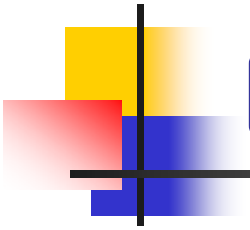
```
line   float sum(float *a, const int n)
 1     {
 2           float s = 0;
 3           for(int i = 0; i < n; i++)
 4                 s += a[i];
 5           return s;
 6     }
```

```
line   float rsum(float *a, const int n)
 1     {
 2           if(n <= 0) return 0;
 3           else return (rsum(a, n-1)+a[n-1]);
 4     }
```



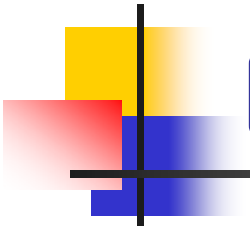
Performance Analysis

- Space complexity
  - fixed part : independent of the characteristics of the inputs and outputs
  - variable part : depends on the instance characteristics
  - S(P) = c+Sp(instance characteristics)
    - S(P):space requirement of program P
    - c : constant



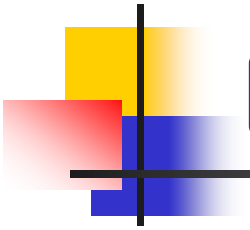
Example [Iterative sum]

- The problem instances are characterized by n
- n is passed by value : 1 word is allocated
- a is the address of a[0], 1 word is needed
- the space is independent of n: Sp(n)=0



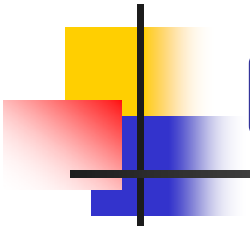
Example [Recursive sum]

- The problem instances are characterized by n
- The depth of recursion depends on n : it is n+1
- Each call requires at least 4 words : n, a, returned value, return address
- The recursion stack space is 4(n+1)



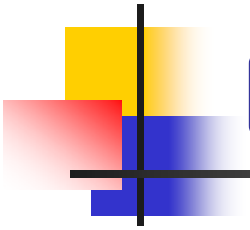
Definition [Big "oh"]

- f(n)=O(g(n)) iff there exist positive constants c and n_0 such that
  - $f(n) \leq cg(n)$ for all n, $n \geq n_0$
- Examples
  - $3n+3 = O(n)$ as $3n+3 \leq 4n$ for all $n \geq 3$
  - $3n+3 = O(n^2)$ as $3n+3 \leq 3n^2$ for $n \geq 2$



Definition [Omega]

- $f(n)=\Omega(g(n))$ iff there exist positive constants c and n_0 such that
  - $f(n) \geq cg(n)$ for all n, $n \geq n_0$
- g(n) is a lower bound
- Examples
  - $3n+2=\Omega(n)$
  - $10n^2+4n+2=\Omega(n^2)$



Definition [Theta]

- $f(n)=\Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0 such that
  - $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$
- g(n) is both an upper and lower bound
- Examples
  - $3n+2=\Theta(n)$
  - $10n^2+4n+2=\Theta(n^2)$
  - $6 \times 2^n+n^2=\Theta(2^n)$

Practical Complexities

- For large n, only programs of small complexity are feasible