RUNNING TIME ANALYSIS

Problem Solving with Computers-II

Performance questions

- How efficient is a particular algorithm?
  - **CPU time usage (Running time complexity)**
  - **Memory usage (Space complexity)**
  - Disk usage
  - Network usage

- Why does this matter?
  - Computers are getting faster, so is this really important?
  - Data sets are getting larger – does this impact running times?

How can we measure time efficiency of algorithms?

- One way is to measure the absolute running time
- Pros? Cons?

```
clock_t t;
t = clock();

//Code under test

t = clock() - t;
```

Which implementation is significantly faster ?

A.

```
double Fib(int n){
    if(n <= 2) return 1;
    return Fib(n-1) + Fib(n-2);
}
```

B.

```
double Fib(int n){
    double *fib = new double[n];
    fib[0] = fib[1] = 1;
    for(int i = 2; i < n; i++){
     fib[i] = fib[i-1] + fib[i-2];
    }
    return fib[n-1];
}
```

C. *Both are almost equally fast*

n	1	2	3	4	5	6	7
Fib(n)	1	1	2	3	5	8	13

A better question: How does the running time grow as a function of input size

```
double Fib(int n){
   if(n <= 2) return 1;
   return Fib(n-1) + Fib(n-2);
}
```

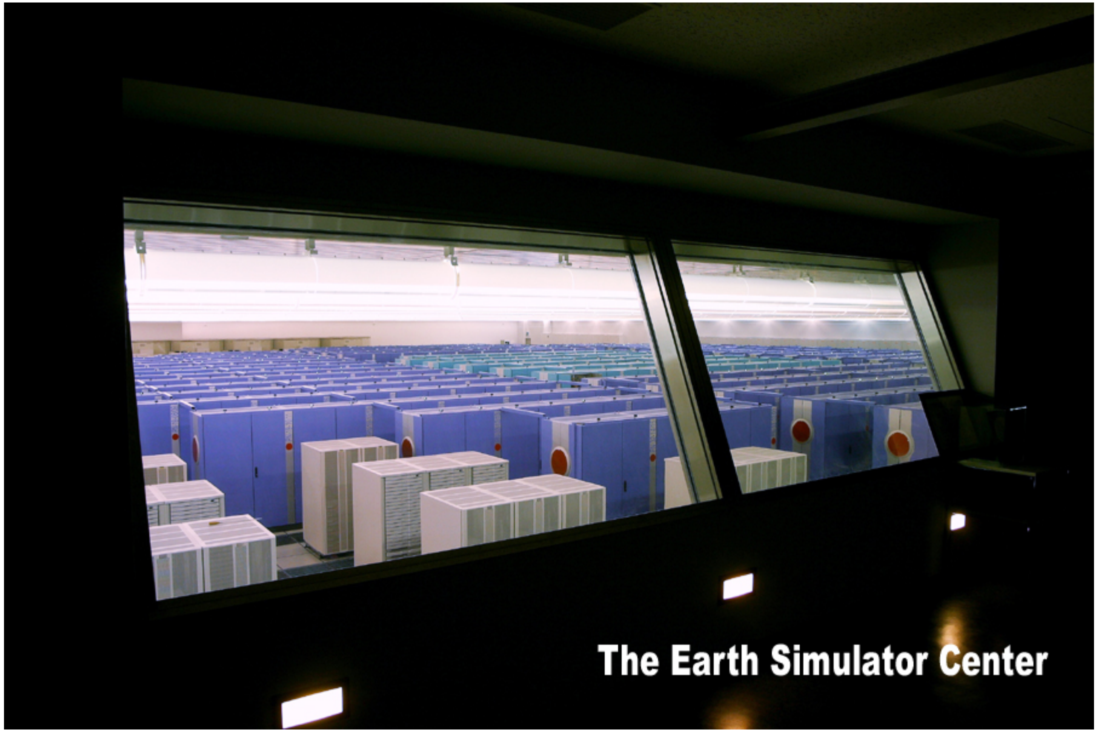
```
double Fib(int n){
    double *fib = new double[n];
    fib[0] = fib[1] = 1;
    for(int i = 2; i < n; i++){
     fib[i] = fib[i-1] + fib[i-2];
    }
    return fib[n-1];
}
```

The “right” question is: How does the running time grow?

E.g. How long does it take to compute Fib(200) recursively?

….let’s say on….

NEC Earth Simulator



Can perform up to 40 trillion operations per second.

The running time of the recursive implementation

The Earth simulator needs 2^{92} seconds for F_{200}.

Time in seconds	Interpretation
2^{10}	17 minutes
2^{20}	12 days
2^{30}	32 years
2^{40}	cave paintings
2^{70}	The big bang!

Let's try calculating F_{200} using the iterative algorithm on my laptop…..

Subgoal 1: Focus on the impact of the algorithm:

Simplify the analysis of running time by ignoring “details” which may be an artifact of the underlying implementation

Subgoal 1: Focus on the impact of the algorithm:

Simplify the analysis of running time by ignoring “details” which may be an artifact of the underlying implementation

- **Subgoal 2: Focus on trends as input size increases (asymptotic behavior):**

How does the running time of an algorithm increases with the size of the input in the limit (for large input sizes)

Subgoal 1: Focus on the impact of the algorithm:

Count operations instead of absolute time!

- Every computer can do some primitive operations in constant time:
  - Data movement (assignment)
  - Control statements (branch, function call, return)
  - Arithmetic and logical operations

```
double Fib(int n){
    double *fib = new double[n];
    fib[0] = fib[1] = 1;
    for(int i = 2; i < n; i++){
     fib[i] = fib[i-1] + fib[i-2];
    }
    return fib[n-1];
}
```

- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm

Subgoal 1: Focus on the impact of the algorithm:

Count operations instead of absolute time!

$T(n)$: Running time of algo for input n

```
double Fib(int n){
  double *fib = new double[n];
  fib[0] = fib[1] = 1;
  for(int i = 2; i < n; i++){
   fib[i] = fib[i-1] + fib[i-2];
  }
  return fib[n-1];
}
```

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

$$= 3 + (n-2)\cdot 8 + 2 = 8n + 5 - 16 = 8n - 11$$

Subgoal 1: Focus on the impact of the algorithm:

Count operations instead of absolute time!

$T(n) = 1 + 2 + (n-2)\cdot 6 = 6n - 9$ $\qquad T'(n) = 8n - 11$

```
procedure Fib(n: positive integer)
 Create an array fib[1..n]        1 step
 fib[1] := 1                 }  2 steps
 fib[2] := 1                 }
 for i := 3 to n:    (n-2)
    fib[i] := fib[i−1] + fib[i−2] 6 steps
 return fib[n]
```

```
double Fib(int n){
    double *fib = new double[n];
    fib[0] = fib[1] = 1;
    for(int i = 2; i < n; i++){
     fib[i] = fib[i−1] + fib[i−2];
    }
    return fib[n−1];
}
```

Can we count number of operations on the pseudo code version of the Fib function?

A. Yes

B. No

Our first goal for analyzing runtime was to focus on the impact of the algorithm. What was our second subgoal?

A. Focus on optimizing the algorithm so that it can be efficient

B. Focus on measuring the time it takes to run the algorithm by time stamping our code.

C. Focus on trends as input size increases (asymptotic behavior)

UCSB CS & CE

SPEED ADVISING

Come speak to your professor about research opportunities, internships, electives, grad shcools, or any topic related to CS and CE at UCSB!

- Research Opportunities
- Internship and Job Advice
- Major Field Electives
- Graduate School Advice
- General Networking

Additionally, **for CS students only**, attending this event can satisfy and complete your Major Elective Approval requirement, which is a requirement for graduation.

Refreshments will be served.

OCT 25
2 PM - 6 PM

HFH 1132

Orders of growth

An **order of growth** is a set of functions whose asymptotic growth behavior is considered equivalent. For example, 2*n*, 100*n* and *n*+1 belong to the same order of growth

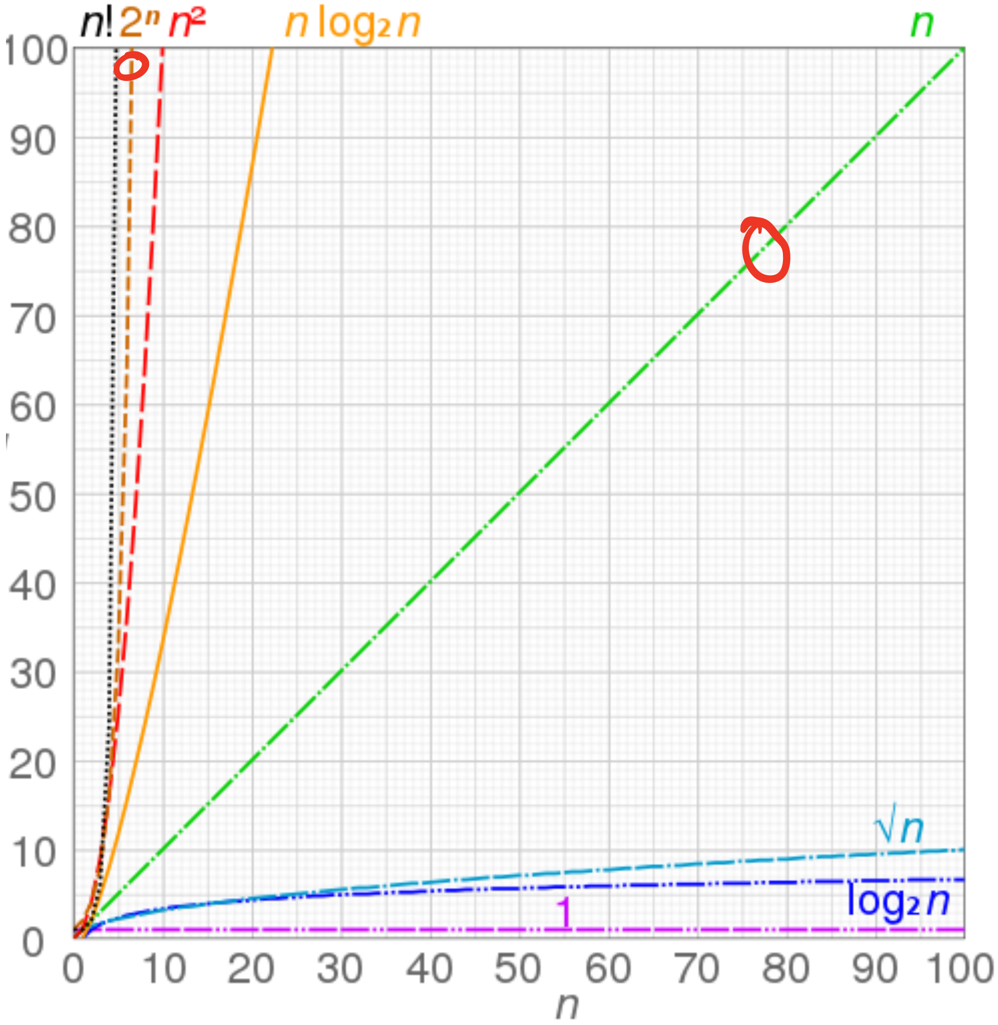
Which of the following functions has a higher order of growth?

A. 50n

B. $2n^2$

$T(n) = 6n - 9$

$T'(n) = n^2 + n\log n + 5$



Big-O notation

- Big-O notation provides an upper bound on the order of growth of a function

$T(n) = 6n-5$

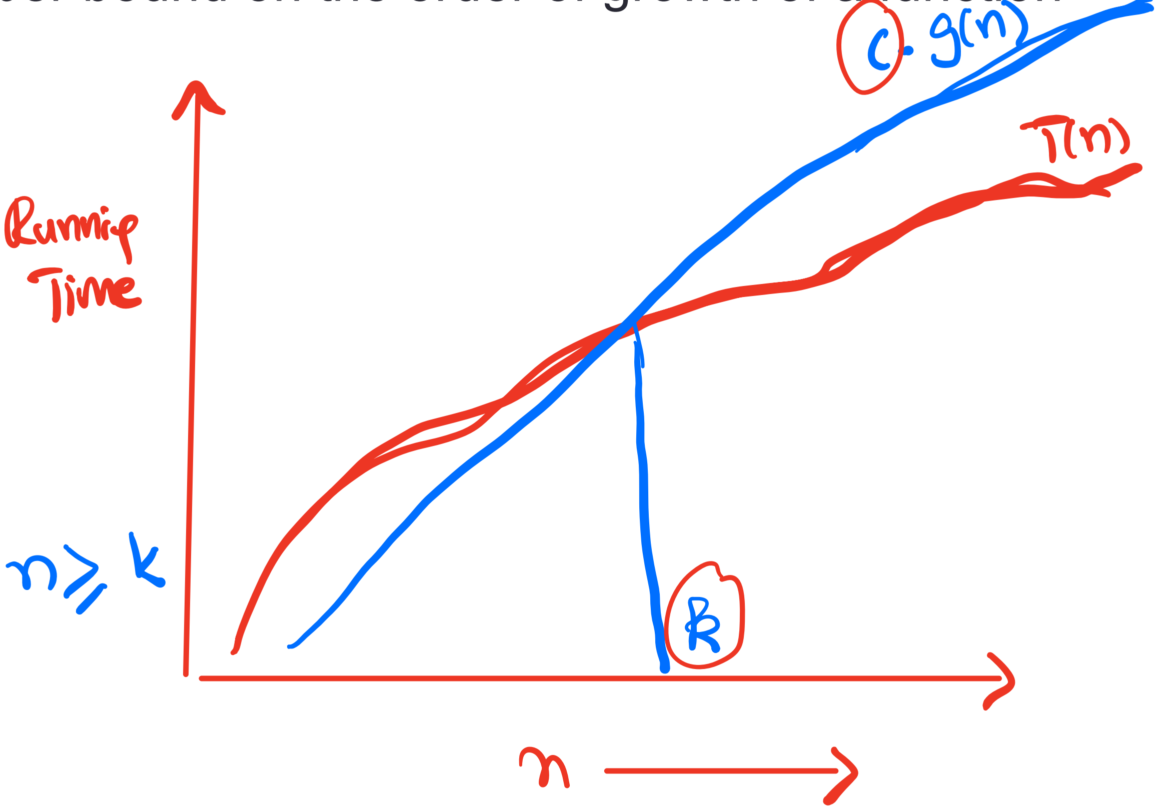
$= O(n)$

Write

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

Means

$T(n) \le c \cdot g(n), \; n \ge k$



$T(n) = 6n - 5$

$\leq 6n \quad , \quad n \geq 1$

$= O(n)$

$T(n) = n^2 + n \log n + 5$

$\leq n^2 + n^2 + 5 \; ,$

$\log n \leq n$

$n \leq 2^n$

for all $n \geq 1$

$\leq n^2 + n^2 + n^2 \quad , \quad n \geq 3$

$= 3n^2$

$T(n) \leq 3 \cdot n^2 \; , \quad n \geq 3$

$= O(n^2)$

Definition of Big-O

f(n) and g(n) map positive integer inputs to positive reals.

We say $f = O(g)$ if there is a constant $c > 0$ and $k > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n >= k$.

$f = O(g)$
means that "f grows no faster than g"

Express in Big-O notation

1. 10000000 = O(1)
2. 3*n = O(n)
3. 6*n-2 = O(n)
4. 15*n + 44 = O(n)
5. 50*n*log(n) = O(n log n)
6. n^2 = $O(n^2)$
7. n^2-6n+9 = $O(n^2)$
8. $3n^2$+4*log(n)+1000 = $O(n^2)$
9. $3^n + n^3$ +log(3*n) = $O(3^n)$

For polynomials, use only leading term, ignore coefficients: linear, quadratic

Common sense rules of Big-O

1. Multiplicative constants can be omitted: $14n^2$ becomes n^2 .
2. n^a dominates n^b if $a > b$: for instance, n^2 dominates n.
3. Any exponential dominates any polynomial: 3^n dominates n^5 (it even dominates 2^n).

Big-O analysis of **iterative** fibonacci

```
procedure Fib(n: positive integer)
 Create an array fib[1..n]
 fib[1] := 1
 fib[2] := 1
 for i := 3 to n:
    fib[i] := fib[i–1] + fib[i–2]
 return fib[n]
```

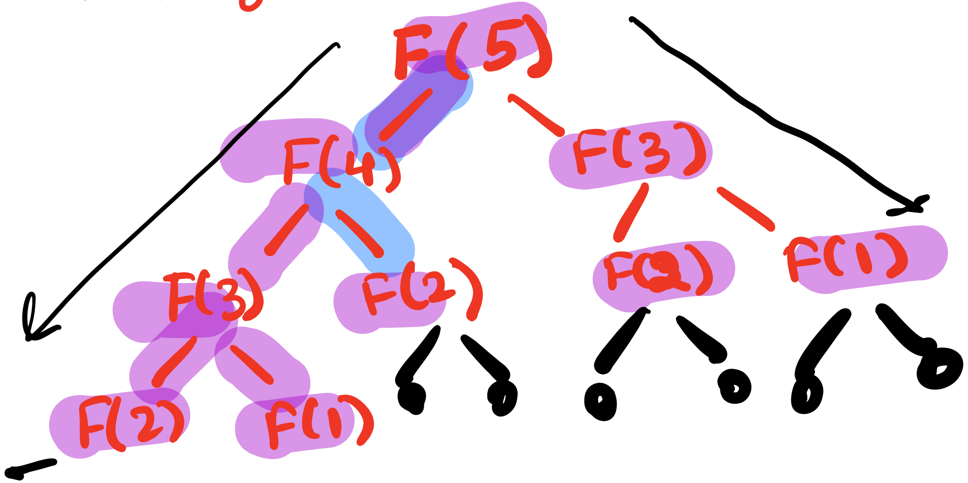
$$T(n) = 6n-9$$

$$= O(n)$$

recursive fibonacci: some observations

```
procedure F(n: a positive integer)
    if(n <= 2) return 1
    return F(n-1) + F(n-2)
```

$T(n) \propto$ # of times $F(\cdot)$ is called for input n



Height of the tree = 3

Full binary tree.

$B(3) = 15$

- Path – a sequence of (zero or more) connected nodes.
- Length of a path - number of edges traversed on the path
- Height of node – Length of the longest path from the node to a leaf node.
- **Height of the tree** - Length of the longest path from the **root** to a leaf node.

Ex. path $P: F(5) - F(4) - F(2)$

length $L(p): 2$

Height of the tree : 3

Types of Binary Trees

Complete Binary Tree: Every level, except possibly the last, is completely filled, and all nodes on the last level are as far left as possible

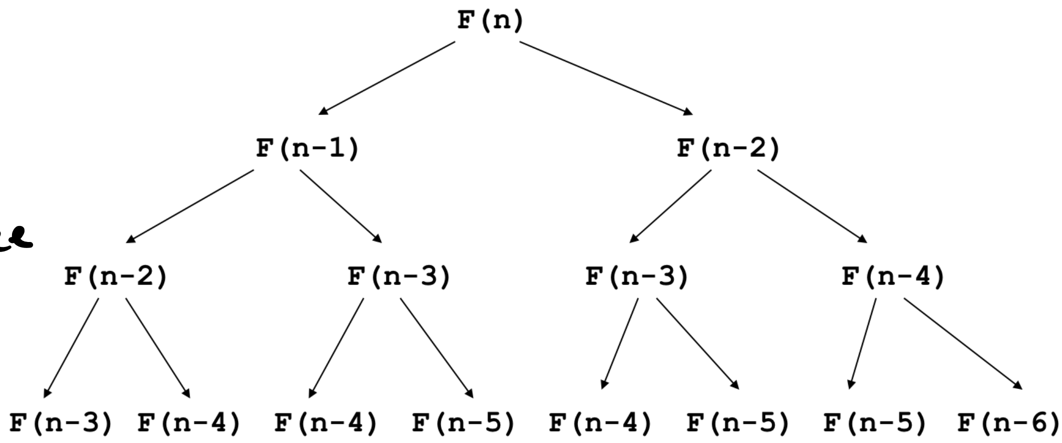
Full Binary Tree: A complete binary tree whose last level is completely filled

Big O analysis of **recursive** Fibonacci

Approach (1)

```
procedure F(n: a positive integer)
    if(n <= 2) return 1
    return F(n-1) + F(n-2)
```

What takes so long? Let's unravel the recursion…



The same subproblems get solved over and over again!

$W(n)$: No. of calls to $F(\cdot)$ for input n

$B(n)$: No. of nodes in a full binary tree of height n

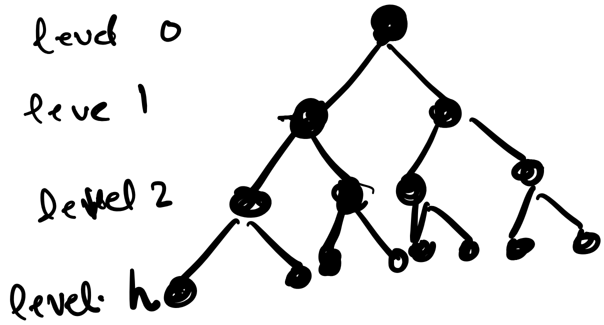
$W(5) = 9$

$B(3) = 15$

In general,

$$W(n) \le B(n-2)$$

We will derive a closed form expression for $B(h)$



For a full binary tree of height h the total number of nodes in the tree, $B(h)$, is given by:

$$B(h) = 2^0 + 2^1 + 2^2 + \dots 2^h$$ (summing the geometric series we get result A)

$$B(h) = 2^{h+1} - 1 \quad \text{— (A)}$$

We observe that the no. of calls to $F(\cdot)$ for input n, denoted by $W(n)$ is bounded by the number of nodes in a full binary tree of height $n-2$

Therefore, $$W(n) \le B(n-2)$$
$$= 2^{n-2+1} - 1 \quad \text{(using result A)}$$
$$= 2^{n-1} - 1$$

$$T(n) = a \cdot W(n) + b$$, for some constants a & b
$$\le a(2^{n-1} - 1) + b$$
$$= O(2^n)$$

Another approach

$T(n)$: Running time of $F(n)$

```
F(n) :
   if n ≤ 2   return 1
   return F(n-1) + F(n-2)
```

$T(1) = 1$

$T(2) = 1$

$T(n) = T(n-1) + T(n-2) + c$, $c = 4$ (This is a recurrence relation)

$\leq T(n-1) + T(n-1) + c$ (Since $T(n-2) \leq T(n-1)$ for $n \geq 2$)

$= 2 \cdot T(n-1) + c$

$\leq 2 \cdot (2 \cdot T(n-2) + c) + c$

$= 2^2 \cdot T(n-2) + 2 \cdot c + c$

$T(n) \leq 2^k \cdot T(n-k) + (2^k - 1) \cdot c$ — (1)

We reach the base case when $n-k = 1$

Substitute for k (in terms of n) in (1), we get

$T(n) \leq 2^{(n-1)} \cdot T(1) + (2^{n-1} - 1) \cdot c$

$= 2^{n-1}(1+c) - c$ (Since $T(1) = 1$)

$= O(2^n)$

```
procedure max(a1,a2, … an: integers)
  max:= a1          c1
  for i:= 2 to n
      if max < ai     c2
          max:= x
return max{max is the greatest element}
```

$$T(n) = c_1 + (n-2)\cdot c_2$$

$$= O(n)$$

What is the Big-O running time of max?

A. $O(n^2)$
B. $O(n)$
C. $O(n/2)$
D. $O(\log n)$
E. None of the above

What is the Big O running time of sum()?

A. $O(n^2)$
B. O(n)
C. O(n/2)
D. O(log n)
E. None of the array

```
/* n is the length of the array*/
int sum(int arr[], int n)
{
	int result=0;
	for(int i=0; i < n; i+=2)
		result+=arr[i];
	return result;
}
```

What is the Big O running time of sum()?

A. $O(n^2)$
B. O(n)
C. O(n/2)
D. O(log n)
E. None of the array

```
/* n is the length of the array*/
int sum(int arr[], int n)
{
        int result = 0;
        for(int i=1; i < n; i=i*2)
                result+=2*arr[i];
         return result;
}
```

c_1

c_2

$T(n) = c_1 + \log n \cdot c_2$

What is the Big O running time of sum()?

A. $O(n^2)$
B. $O(n)$
C. $O(n/2)$
D. $O(\log n)$
E. None of the array

```
/* n is the length of the array*/
int sum(int arr[], int n)
{
        int result = 0;
        for(int i=0; i < n; i=i+2)
                result+=arr[i];
        for(int i=1; i < n; i=i*2)
                result+=2*arr[i];
        return result;
}
```

O(1)

O(n)

O(log n)

$$T(n) = O(1) + O(n) + O(\log n) = O(n)$$

Next time

- Running time analysis : best case and worst case
- Running time analysis of Binary Search Trees

References:
https://cseweb.ucsd.edu/classes/wi10/cse91/resources/algorithms.ppt
http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf