Basic Concepts

Chapter 1

- int i=10, j=20;
- i = j;

- int i, *pi;
- pi = &i;
- i=10;
- *pi=10;

- int i,j, *pi, *pj;
- i=10; j=20;
- pi = &i;
- pj = &j;
- *pi=*pj;

- if (pi==NULL)
- if (!pi)

Contents

performance analysis
vs.
performance measurement

- [machine independent] time and space estimate

- [machine dependent] running times

Performance Analysis (1)

- 성능/효율 분석
- One of the goals of this lecture is to develop your skills for making evaluative judgments about programs.
  - Does the program efficiently use primary and secondary **storage**?
  - Is the program's running **time** acceptable for the task?

Performance Analysis (2)

- program에 대해 machine과 독립적인 time, space를 분석하는 것
  - Time complexity:

    amount of computation time that a program needs to run to completion

  - Space complexity:

    amount of memory that a program needs to run to completion

Which has better performance? (1)

```
float sum(float list[], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
}
```

Program 1.11: Iterative function for summing a list of numbers

```
float rsum(float list[], int n)
{
   if (n) return rsum(list,n-1) + list[n-1];
   return 0;
}
```

Program 1.12: Recursive function for summing a list of numbers

```c
float sum(float list[], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
}
```

Program 1.11: Iterative function for summing a list of numbers

```
float rsum(float list[], int n)
{
   if (n) return rsum(list,n-1) + list[n-1];
   return 0;
}
```

Program 1.12: Recursive function for summing a list of numbers

n이 10

Which has better performance? (2)

- Fibonacci 수열: F(0)=1, F(1)=1, F(i)=F(i-1)+F(i-2) for i>1
  - Iterative version
  - Recursive version
- 1부터 n 까지의 정수 합 계산 방법
  - Iterative version
  - Constant time function

Some Uses Of Performance Analysis

- Determine practicality of algorithm
- Predict run time on large instance
- Compare two algorithms that have different asymptotic complexity (점근적 복잡도)
  - e.g., $O(n)$ and $O(n^2)$

Program Complexity

- 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 (연역적 평가)
  - performance measurement : a posteriori testing (귀납적 테스팅)

Space Complexity (1)

- Fixed part : program의 input/output size에 무관한 space
  - Instruction space
  - space for simple variables
  - fixed-size structured variables
  - constants

Space Complexity (2)

- Variable part : depends on the instance characteristics (ex: dynamic memory allocation, using stacks for recursion,..)
- S(P) = c+Sp(n)
  - S(P):space requirement of program P
  - c : constant (for fixed space requirements)
  - n : Instance characteristics (ex: I/O size, number)

Space Complexity : Example(1)

```
float abc(float a, float b, float c)
{
   return a+b+b*c+(a+b-c)/(a+b)+4.00;
}
```

Program 1.10: Simple arithmetic function

$S_{abc}(n) = 0$

array가 인자로 전달되는 방식

```
float sum(float list[], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
}
```

Program 1.11: Iterative function for summing a list of numbers

$S_{sum}(n) =$

Space Complexity : Example(2)

```
float rsum(float list[], int n)
{
   if (n) return rsum(list,n-1) + list[n-1];
   return 0;
}
```

Program 1.12: Recursive function for summing a list of numbers

Type	Name	Number of bytes
parameter: array pointer	*list*[]	4
parameter: integer	*n*	4
return address: (used internally)		4
TOTAL per recursive call		12

If n is MAX_SIZE

S_{rsum}(MAX_SIZE) = 12 * (MAX_SIZE+1);

Time complexity

- $T(P) = c + Tp(n)$
  - T(P) : time taken by program P
  - c : compile time
  - Tp : run time
  - n : instance characteristics

컴파일은 한번 수행. 따라서 Tp가 중요함.

Program Steps

- We could count the number of operations the program performs to analyze the time requirements
  - Machine-independent estimate
  - We must know how to divide the program into distinct steps
- Definition: *A program step*
  - A syntactically or semantically meaningful segment of a program whose execution time is independent of the instance characteristics
    - A = 2; // 1step
    - A = 3*b + A/200 – c/d/e/f; // 1 step

Determining the number of steps : examples

Only need to worry about executable statements

```
float sum(float list[], int n)
{
   float tempsum = 0;                /* for assignment */
   int i;
   for (i = 0; i < n; i++)   {
                                     /* for the for loop */
     tempsum += list[i]; count++;   /* for assignment */
   }
                /* last execution of for */
                /* for return */  return tempsum;
}
```

Program 1.13: Program 1.11 with count statements

Number of steps?

Using the variable 'count'

```
float sum(float list[], int n)
{
  float tempsum = 0;  count++; /* for assignment */
  int i;
  for (i = 0; i < n; i++)  {
    count++;                    /* for the for loop */
    tempsum += list[i]; count++;  /* for assignment */
  }
  count++; /* last execution of for */
  count++; /* for return */  return tempsum;
}
```

Program 1.13: Program 1.11 with count statements

```
float sum(float list[], int n)
{
  float tempsum = 0;
  int i;
  for (i = 0; i < n; i++)
    count += 2;
  count +=3;
  return 0;
}
```

Program 1.14: Simplified version of Program 1.13

steps

Determining the number of steps : examples

```
float rsum(float list[], int n)
{
   if (n) return rsum(list,n-1) + list[n-1];
   return 0;
}
```

Program 1.12: Recursive function for summing a list of numbers

```
float rsum(float list[], int n)
{
   count++;      /* for if conditional */
   if (n) {
      count++;  /* for return and rsum invocation */
      return rsum(list,n-1) + list[n-1];
   }
   count++;
   return list[0];
}
```

Program 1.15: Program 1.12 with count statements added

Number of steps?

Determining the number of steps : examples

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                  int c[][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++)
      for (j = 0; j < cols; j++)
         c[i][j] = a[i][j] + b[i][j];
}
```

Program 1.16: Matrix addition

Number of steps?

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                  int c[][MAX_SIZE], int rows, int cols)
{
  int i, j;
  for (i = 0; i < rows; i++) {
    count++;  /* for i for loop */
    for (j = 0; j < cols; j++) {
      count++; /* for j for loop */
      c[i][j] = a[i][j] + b[i][j];
      count++; /*  for assignment statement */
    }
    count++; /* last time of j for loop */
  }
  count++; /* last time of i for loop */
}
```

Program 1.17: Matrix addition with count statements

Determining the number of steps : <u>step count table</u>

statement	steps	freq	total steps
float sum (float list[], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for (i = 0; i < n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
total			2n+3

Determining the number of steps : step count table

statement	steps	freq	total steps
float rsum(float list[], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1) + list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
total			2n+2

Determining the number of steps : step count table

Statement	s/e	Frequency	Total Steps
void add(int a[][MAX_SIZE] $\cdots$)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i=0; i<rows; i++)	1	$rows+1$	$rows+1$
for (j = 0; j < cols; j++)	1	$rows \cdot (cols+1)$	$rows \cdot cols + rows$
c[i][j] = a[i][j] + b[i][j];	1	$rows \cdot cols$	$rows \cdot cols$
}	0	0	0
Total			$2rows \cdot cols + 2rows+1$

Asymptotic notation (O, Ω, Θ) (1)

- Determining the exact step count is very difficult
- The notion of a step is inexact – not so useful for comparative purposes.
  - 80n, 85n, 75n
  - Exceptional case: 10000n+10 vs. 3n+3

We need to be able to make meaningful statements about the time and space complexities of a program.

Asymptotic notation (O, Ω, Θ) (2)

- When c_1, c_2, c_3 are nonnegative constants, $c_1\, n^2 + c_2 n > c_3 n$ for sufficiently large value of n

  $c_1 = 1,\ c_2 = 2,\ c_3 = 100$

  $c_1 n^2 + c_2 n \le c_3 n,\quad n \le 98$

  $c_1 n^2 + c_2 n > c_3 n,\quad n > 98$

- break even point (손익분기점) : n = 98

Definition [Big "oh"] (1)

- *$f(n)=O(g(n))$ (read as "f of n <u>is</u> **big oh** of g of n")*
  iff $\exists c, n_0 > 0$, s.t. $f(n) \leq cg(n)\ \ \forall n,\ n \geq n_0$

- *$f(n) = O(g(n))$ ('=' means 'is' not 'equal')*
  - $\forall n,\ n \geq n_0$, g(n) is upper bound on f(n)
  - g(n) should be the smallest value

Definition [Big "oh"] (2)

- e.g:

  $3n+3 = O(n)$ as $3n+3 \leq 4n$ for all $n \geq 3$

  $3n+3 = O(n^2)$ as $3n+3 \leq 3n^2$ for $n \geq 2$

  $10n^2 + 4n + 2 =$

  $6*2^n + n^2 =$

cubic

constant

- $O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n)$

linear

quadratic

exponential

Definition [Omega]

f of n is omega of g of n

$f(n)=\Omega(g(n))$ *iff*

$\exists c, n_0 > 0$, *s.t.* $f(n) \geq cg(n) \ \ \forall n, n \geq n_0$

- g(n) is a lower bound
- g(n) should be the largest value

e.g: $3n+2=\Omega(n)$ $3n + 2 >= 3n$ for $n>=1$

$10n^2+4n+2=\Omega(n^2)$ $10n^2+4n+2>=n^2$ for $n>=1$

$6*2^n + n^2 =$

$6*2^n+n^2>=2^n$ for $n>=1$

Definition [Theta]

- *$f(n)=\Theta(g(n))$ iff $\exists c_1, c_2, n_0 > 0$, s.t. $c_1 g(n) \le f(n) \le c_2 g(n)$ $\forall n, n \ge n_0$*

  f of n is theta of g of n

- g(n) is both an upper and lower bound

  e.g: $3n+2=\Theta(n)$ $3n \le 3n+2 \le 4n$ for all $n \ge 2$

  $10n^2+4n+2=\Theta(n^2)$

  $6 \times 2^n+n^2=\Theta(2^n)$

  $2*n*m + 2*n + 1 =$

* $f(n) = \Theta(g(n))$ iff g(n) is both an upper and lower bound on f(n)
* coefficient of g(n) is 1 !! – We do not write $\Theta(32n)$ but $\Theta(n)$.

Asymptotic Complexity: Examples

Statement	Asymptotic complexity
void add(int a[][MAX_SIZE] $\cdots$)	0
{	0
int i, j;	0
for (i=0; i<rows; i++)	$\Theta(rows)$
for (j = 0; j < cols; j++)	$\Theta(rows.cols)$
c[i][j] = a[i][j] + b[i][j];	$\Theta(rows.cols)$
}	0
Total	$\Theta(rows.cols)$

Practical Complexities (1)

- time complexity
  - f(instance characteristic of P)
  - instance char.이 변할 때 time complexity의 변화량 추정
  - 동일기능 프로그램 P, Q의 time complexity 비교
- 충분히 큰 instance char.에 대해 비교해야 함.

Practical Complexities (2)

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

Figure 1.7: Function values

	$f(n)$						
n	n	$n\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 h	1 ms
30	.03 μ	.15 μ	.9 μ	27 μ	810 μ	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121 d	18 m
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 y	13 d
100	.10 μs	.66 μs	10 μs	1 ms	100 ms	3171 y	$4*10^{13}$ y
10^3	1 μs	9.96 μs	1 ms	1 s	16.67 m	$3.17*10^{13}$ y	$32*10^{283}$ y
10^4	10 μs	130 μs	100 ms	16.67 m	115.7 d	$3.17*10^{23}$ y	
10^5	100 μs	1.66 ms	10 s	11.57 d	3171 y	$3.17*10^{33}$ y	
10^6	1 ms	19.92 ms	16.67 m	31.71 y	$3.17*10^7$ y	$3.17*10^{43}$ y	

μs = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
s = seconds; m = minutes; h = hours; d = days; y = years

Figure 1.9: Times on a 1-billion-steps-per-second computer

Practical Complexities (3)

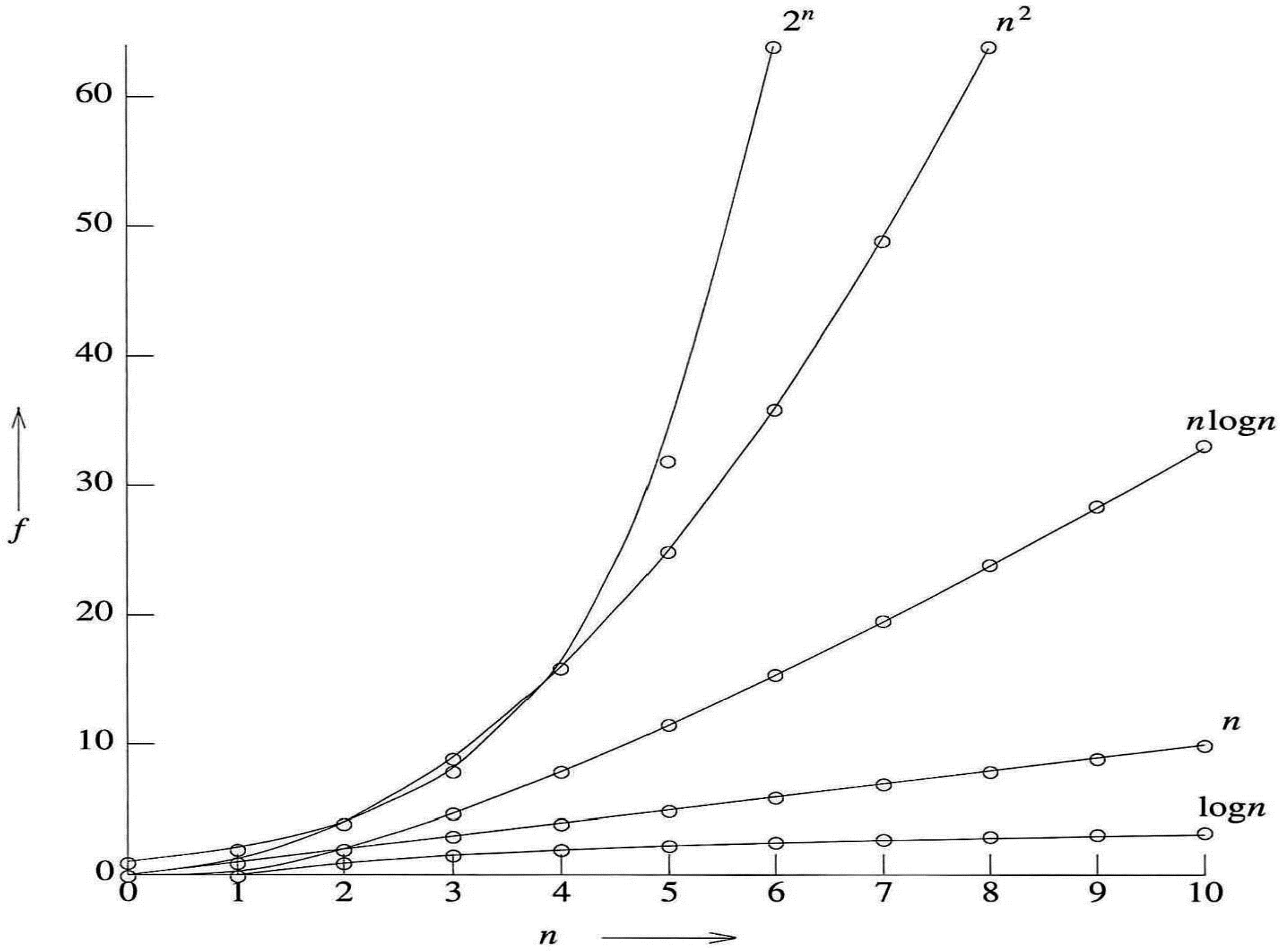

Figure 1.8 Plot of function values

Performance Measurement

- Measure actual time on an actual computer.
- How does the algorithm execute on our machine?
  - analysis보다 measurement가 필요
- 2가지 측정도구 사용: clock, time

Performance Measurement Needs

- Data to use for measurement
  - worst-case data
  - best-case data
  - average-case data

- Timing mechanism --- clock

Event timing In C

- Use clock() function or time() function in the C standard library.
- #include<time.h>

	Method 1	Method 2
Start timing	start = clock();	start = time(NULL);
Stop timing	stop = clock();	stop = time(NULL);
Type returned	clock_t	time_t
Result in seconds	duration = ((double) (stop−start)) / CLOCKS_PER_SEC;	duration = (double) difftime(stop,start);

selection sort

인덱스	<u>0</u>	1	2	3	4	5
값	**5**	**8**	**1**	**7**	**3**	**9**

```
void sort(int list[],int n)
{
   int i, j, min, temp;
   for (i = 0; i < n-1; i++)  {
      min = i;
      for (j = i+1; j < n; j++)
         if (list[j] < list[min])
            min = j;
      SWAP(list[i],list[min],temp);
   }
}
```

Event timing in C : a sample program

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
   int i, n, step = 10;
   int a[MAX_SIZE];
   double duration;
   clock_t start;

   /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
   printf("    n     time\n");
   for (n = 0; n <= 1000; n += step)
   {/* get time for size n */

      /* initialize with worst-case data */
      for (i = 0; i < n; i++)
         a[i] = n - i;

      start = clock( );
      sort(a, n);
      duration = ((double) (clock() - start))
                              / CLOCKS_PER_SEC;
      printf("%6d   %f\n", n, duration);
      if (n == 100) step = 100;
   }
}
```

Program 1.24: First timing program for selection sort

n	time
0	0.000000
10	0.000000
20	0.000000
30	0.000000
40	0.000000
50	0.000000
60	0.000000
70	0.000000
80	0.000000
90	0.000000
100	0.000000
200	0.000000
300	0.000000
400	0.000000
500	0.000000
600	0.000000
700	0.000000
800	0.001000
900	0.001000
1000	0.001000

```c
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
   int i, n, step = 10;
   int a[MAX_SIZE];
   double duration;

   /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
   printf("    n    repetitions     time\n");
   for (n = 0; n <= 1000; n += step)
   {
      /* get time for size n */
      long repetitions = 0;
      clock_t start = clock( );
      do
      {
         repetitions++;

         /* initialize with worst-case data */
         for (i = 0; i < n; i++)
            a[i] = n - i;

         sort(a, n);
      } while (clock( ) - start < 1000);
            /* repeat until enough time has elapsed */

      duration = ((double) (clock() - start))
                              / CLOCKS_PER_SEC;
      duration /= repetitions;
      printf("%6d  %9d   %f\n", n, repetitions, duration);
      if (n == 100) step = 100;
   }
}
```

Program 1.25: More accurate timing program for selection sort

n	repetitions	time
0	8690714	0.000000
10	2370915	0.000000
20	604948	0.000002
30	329505	0.000003
40	205605	0.000005
50	145353	0.000007
60	110206	0.000009
70	85037	0.000012
80	65751	0.000015
90	54012	0.000019
100	44058	0.000023
200	12582	0.000079
300	5780	0.000173
400	3344	0.000299
500	2096	0.000477
600	1516	0.000660
700	1106	0.000904
800	852	0.001174
900	681	0.001468
1000	550	0.001818

Iterative binary search

Left Right

1	5	9	14	23	36	41	55	73	85	91

```
int binsearch(int list[], int searchnum, int left,
                                          int right)
{/* search list[0] <= list[1] <=  · · ·  <= list[n-1] for
 searchnum. Return its position if found. Otherwise
 return -1 */
   int  middle;
   while (left <= right)  {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: left = middle + 1;
                  break;
         case 0 : return middle;
         case 1 : right = middle - 1;
      }
   }
   return -1;
}
```

Program 1.7: Searching an ordered list

Iterative binary search

Left ... Right

1	5	9	14	23	36	41	55	73	85	91

```
int binsearch(int list[], int searchnum, int left,
                                          int right)
{/* search list[0] <= list[1] <=  · · ·  <= list[n-1] for
 searchnum. Return its position if found. Otherwise
 return -1 */
   int  middle;
   while (left <= right)  {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: left = middle + 1;
                  break;
         case 0 : return middle;
         case 1 : right = middle - 1;
      }
   }
   return -1;
}
```

Program 1.7: Searching an ordered list

Recursive binary search (1)

- 리스트 A[1, n]에서 v를 찾는 문제
  - A[1, middle]과 A[middle+1, n]에서 v를 찾는 문제로 나뉘며, 나뉜 각 리스트에서도 동일 알고리즘을 적용할 수 있다.
  - if list[middle]<searchnum:
    - binsearch(list, searchnum, middle+1, right)
  - else if list[middle]>searchnum:
    - binsearch(list, searchnum, left, middle-1)

Recursive binary search (2)

Left Right

1	5	9	14	23	36	41	55	73	85	91

```
int binsearch(int list[], int searchnum, int left,
                                                int right)
{/* search list[0] <= list[1] <= ... <= list[n-1] for
    searchnum. Return its position if found. Otherwise
    return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
}
```

Program 1.8: Recursive implementation of binary search

Recurrence equation

- Used for computing time complexity of recursive functions
  - recursive binary search
  - rsum function
  - Fibonacci 수열 (F(0)=1, F(1)=1, F(i)=F(i-1)+F(i-2), i>1)

ex1. rsum function

- Recurrence equation
  - $T(n) = T(n-1)+1$
  - $T(0)=1$
- Solve $T(n)$

$$
\begin{aligned}
T(n) &= T(n-1) + 1 \\
&= (T(n-2) + 1) + 1 \\
&= ((T(n-3) + 1) + 1) + 1 \\
&\cdots \\
&= (\ldots((T(0) \underbrace{+ 1) + 1)\ldots) + 1}_{n} = n+1 = O(n)
\end{aligned}
$$

ex2. Binary search function

- Recurrence equation (assume that $n = 2^k$)
  - $T(n) = T(n/2) + 1$
  - $T(1) = 1$
- Solve T(n), and compute big-oh function

$$
\begin{aligned}
T(n) &= T(n/2) + 1 \\
&= (T(n/2^2) + 1) + 1 \\
&= ((T(n/2^3) + 1) + 1) + 1 \\
&\ldots \\
&= (\ldots((T(n/2^k) \underbrace{+ 1) + 1) \ldots) + 1}_{k} = 1+k = 1+\log_2 n \\
&= O(\log_2 n)
\end{aligned}
$$

Summary

- We learned
  - Performance analysis
    - Asymptotic notation
    - Recurrence equation
  - Performance measurement
    - Clocking
- Those will be used throughout this lecture
  - Both for theoretical exams and for program assignments