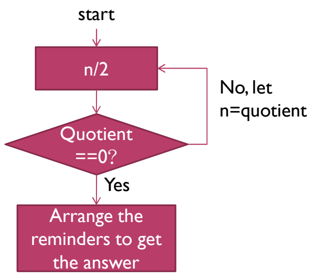
Algorithm

Review what we had learned before

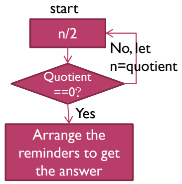
- Algorithm: A finite sequence of instructions to describe a systematical method to solve a problem
- We can represent the 'decimal-to-binary' algorithm by a flow chart
  - It has starting point [step 1]
  - Step 2 is a condition statement
  - Step1+step 2 is a loop statement
  - The problem size is shrunk after each loop
  - The loop can be terminated [when quotient ==0]


start
n/2
Quotient ==0?
No, let n=quotient
Yes
Arrange the reminders to get the answer

十進位轉二進位演算法 Three algorithms

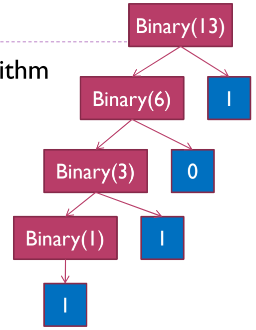
- Problem: converting n_d to m_b.
- Algorithm 1: as mentioned in the last class
- Algorithm 2:
  - $m_b = 0$
  - For i = 1 to n_d
    - $m_b = m_b + 1$
- Algorithm 3:
  - $m_b = 0$
  - While n_d is not 0
    - Find $2^k \le n_d < 2^{k+1}$ → What is the cost of finding k?
    - $m_b = m_b + 2^k$ → The cost of m_b+2^k is just putting 1 at the kth position of m_b
    - $n_d = n_d - 2^k$ → What is the cost of $n_d = n_d - 2^k$?

What is the cost of $m_b = m_b+1$?


start
n/2
Quotient ==0?
No, let n=quotient
Yes
Arrange the reminders to get the answer

Recursive algorithm

- The first algorithm is a recursive algorithm
  - To answer what the binary of 13_d is, you need to answer what the binary of 6_d is.
  - To answer what the binary of 6_d is, you need to answer what the binary of 3_d is.
  - To answer what the binary of 3_d is, you need to answer what the binary of 1_d is.
- Recursive algorithm
  - Turn a big problem into one or several smaller subproblems
  - Each subproblem is identical to the original one except size
    - Thus, they can be solved by the same method.
  - Need a termination condition.


Binary(13)
Binary(6)
1
Binary(3)
0
Binary(1)
1
1

Algorithm

- An effective method for solving a problem using a finite sequence of instructions.
  - It need be able to solve the problem. (correctness)
  - It can be represented by a finite number of instructions.
    - Each instruction must be achievable by computers
    - Assignment, if-then-else statement, loop statement,
  - The more effective, the better algorithm is.
    - How to measure the "efficiency"?
- Outline
  - Sorting problem
    - Correctness, efficiency, recursion

Sorting problem

- Given N numbers, arrange them in the ascending order.
- Algorithm: (in ascending order)
  - Find the smallest element from the list
  - Recursively sort the rest
- In the way that computer can do it

```
void SelectionSort(int n, int a[]){
   if (n==1) return;
   int index = FindSmallest(start, end, a[]);
   Swap(a[0], a[index]);
   SelectionSort(n-1, a[1:n-1]);
}
```

How to prove the correctness?

- Using Induction
  - For n=1, the smallest number is the only number. Therefore, it is sorted.
  - Assume for n=k, the SelectionSort can sort k numbers correctly.
  - For n=k+1, the SelectionSort first finds the smallest element and moves it to a[0], and then sorts the rest k elements.
  - Since the SelectionSort can sort k elements correctly, and the a[0] is smaller than or equal to other k elements, the output array contains the sorted elements in the ascending order.
  - By induction, the SelectionSort is correct.

How efficient is this algorithm?

- How many data comparisons is needed?
  - N(N-1)/2 inside the FindSmallest
  - N for checking N==1
- How many data movements is needed?
  - N(N-1) for the FindSmallest
  - N-1 for Swap
- How many times SelectionSort is called?
  - N-1 times
- If N is doubled, what will 1 and 2 be changed?
  - They will be quadrupled (4X)
  - Number of calls for SelectionSort will be doubled.

Big-theta notation

- Which one is better? $3000*N+9999$ or $0.1*N^2$
- We say $f(x)=\Theta(g(x))$ if $\exists$ $M_1, M_2, k > 0$ and for all $x>k$,
  $M_1 g(x) \leqq f(x) \leqq M_2 g(x)$
  - $f(n) = 3000*n+9999 = \Theta(n)$
  - $f(n) = 0.1*n^2 = (n^2)$

Running time and time complexity

- Suppose your CPU has clock rate 3G HZ (3×10^9) and each operation only takes 1 clock cycle to finish.

Time complexity	N=50	N=51	N=100	$N=10^6$	$N=10^{12}$
$\Theta(1)$	100 s	100 s	100 s	100 s	100 s
$\Theta(\log_2 N)$	$1.88*10^{-9}$ s	$1.89*10^{-9}$ s	$2.21*10^{-9}$ s	$2.21*10^{-9}$ s	$1.3*10^{-8}$ s
$\Theta(N^{1/2})$	$2.36*10^{-9}$ s	$2.38*10^{-9}$ s	$3.33*10^{-9}$ s	$3.33*10^{-7}$ s	$3.33*10^{-5}$ s
$\Theta(N)$	$1.666*10^{-8}$ s	$1.7*10^{-8}$ s	$3.33*10^{-8}$ s	$3.33*10^{-4}$ s	5.56 minutes
$\Theta(N*\log_2 N)$	$9.4*10^{-8}$ s	$9.6*10^{-8}$ s	$2.21*10^{-7}$ s	$6.64*10^{-3}$ s	110 minutes
$\Theta(N^2)$	$8.33*10^{-7}$ s	$8.67*10^{-7}$ s	$3.33*10^{-6}$ s	5.56minutes	10^7 year
$\Theta(2^N)$	4.34 days	8.69 days	$\sim10^{13}$ years		
$\Theta(N!)$	$\sim10^{47}$ years	$\sim10^{49}$ years	$\sim10^{140}$ years		

Can we do better?

- If N is halved, what will the time complexity be?
  - Assume the number of operations for the SelectionSort is
    $T(N) = 3N(N-1) + 3N = 3N^2$
  - Sorting two N/2 numbers takes $T(N/2)+T(N/2)=3/2N^2$
- How to merge two sorted sequences?

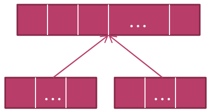
How many comparisons and data data movement are required?

After each comparison, an element will be moved to the new array.

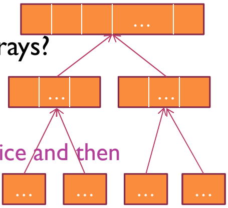
N comparison+data movements to merge two sorted arrays

Two new algorithms

- We can do the sort as follows
  1. Divide the N elements into two N/2 arrays
  2. Use SelectionSort to sort two N/2 arrays
  3. Merge two sorted arrays



- The time complexity for the above algorithm is $3/2N^2+N$
  - Better than the original one $3N^2$.
- What if we divide the data to four N/4 arrays?
  - SelectionSort N/4 data takes ${}^3/_{16}N^2$
  - There are four N/4 arrays to sort ➔ $3/4N^2$
  - When merging, we merge two N/4 arrays twice and then merge two N/2 arrays ➔ 2N
  - The total time of the algorithm is $3/4N^2 +2N$



MergeSort

- Although both algorithm improves the original SelectionSort, the time complexity of them is still $\Theta(N^2)$
  - When N is doubled, the time for them is quadrupled.
- What if we apply the splitting and merging recursively?
  1. Divide the N elements into two N/2 arrays
  2. Use MergeSort to sort two N/2 arrays
  3. Merge two sorted arrays
- What is the time complexity of MergeSort?
  - Let T(N) be the time complexity of MergeSort

    $T(N) = 2T(N/2) + 2N$
  - Let $T(1) = 1$. What is T(N)?

Pictorial view of T(N)

- We can arrange the MergeSorts into layers
  - The MergeSorts in each layer have the same number of data
  - There are 2^k MergeSorts in Layer k.
  - The number of layers is
  - Merging 2^k segments into 2^{k-1} segments takes
  - The time complexity of MergeSort is $N*\log_2 N$
  - When N is doubled, $\log_2 2N = \log_2 N+1$.