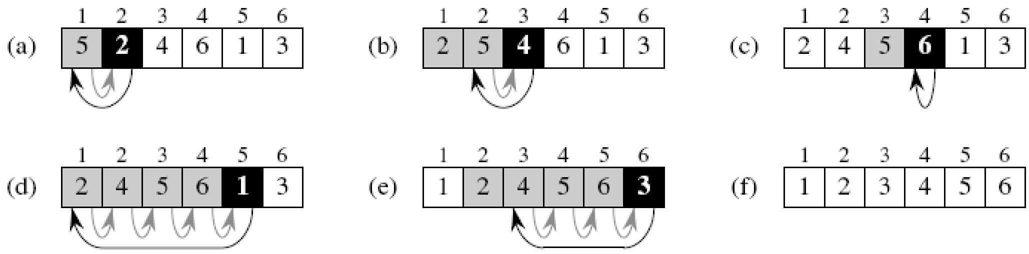
Week 1 lecture 2

Introduction to algorithms

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i − 1
8          A[i + 1] ← key
```



Correctness proof

We state these properties of $A[1\,..\,j-1]$ formally as a ***loop invariant***:

> At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1\,..\,j-1]$ consists of the elements originally in $A[1\,..\,j-1]$ but in sorted order.

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

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.

Analysing insertion sort running time

	INSERTION-SORT(A)	*cost*	*times*
1	**for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2	**do** *key* $\leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1) .$$

Best case running time, array is already sorted.

case occurs if the array is already sorted. For each $j = 2, 3, \ldots, n$, we then find that $A[i] \le key$ in line 5 when i has its initial value of $j-1$. Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

an + b, linear function of n

Worst case running time, array is reverse sorted order.

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1\,..\,j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots, n$. Noting that

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\ .
\end{aligned}
$$

$a n^2 + bn + c$, Quadratic Function of n

Worst and average case analysis

Order of growth, Rate of growth

Worst-case running time $\Theta(n^2)$

Designing algorithms

Insertion sort uses

incremental approach: sort A[1..j-1] then insert A[j] to yield sorted A[1..j]

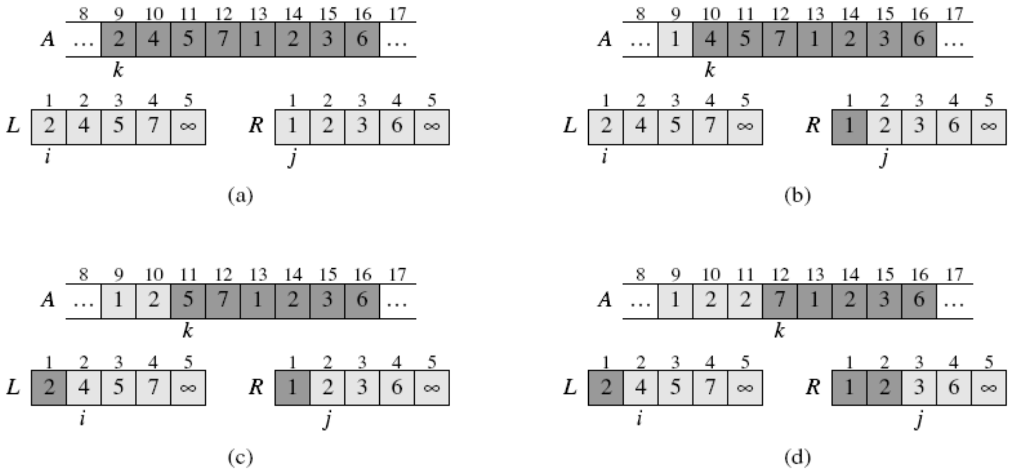
Divide-and-conquer

Divide the problem into a number of subproblems.
Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
Combine the solutions to the subproblems into the solution for the original problem.

Merge sort

Divide: Divide the *n*-element sequence to be sorted into two subsequences of $n/2$ elements each.
Conquer: Sort the two subsequences recursively using merge sort.
Combine: Merge the two sorted subsequences to produce the sorted answer.

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
      do L[i] ← A[p + i − 1]
for j ← 1 to n2
      do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
      do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```



MERGE procedure takes time Θ (n), where $n = r - p + 1$

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

MERGE-SORT(*A,* 1*, length*[*A*]*)*,

its running time can often be described by a ***recurrence equation*** or ***recurrence***,

division of the problem yields *a* subproblems, each of which is 1/*b* the size of the original. If we take *D(n)* time to divide the problem into subproblems and *C(n)* time to combine the solutions to the subproblems into the solution to the original problem.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c , \\ aT(n/b) + D(n) + C(n) & \text{otherwise} . \end{cases}$$

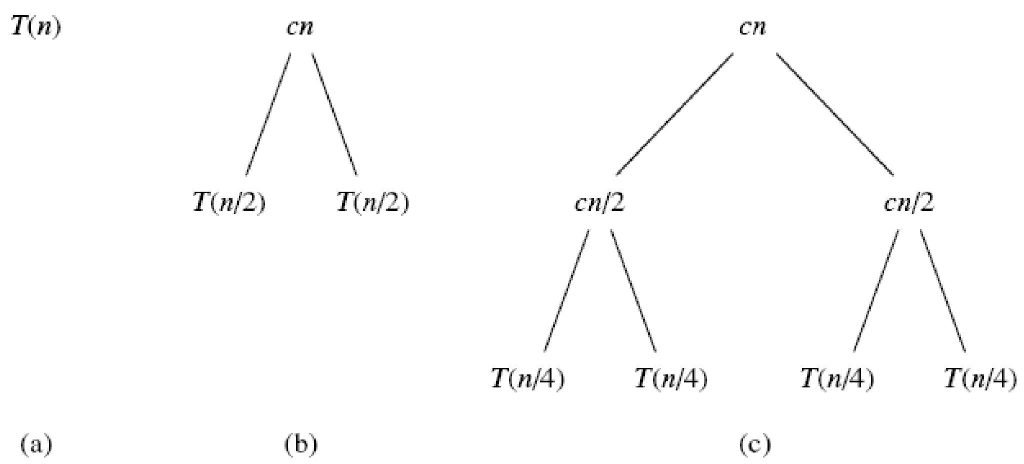
Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.
Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
Combine: We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, so $C(n) = \Theta(n)$.

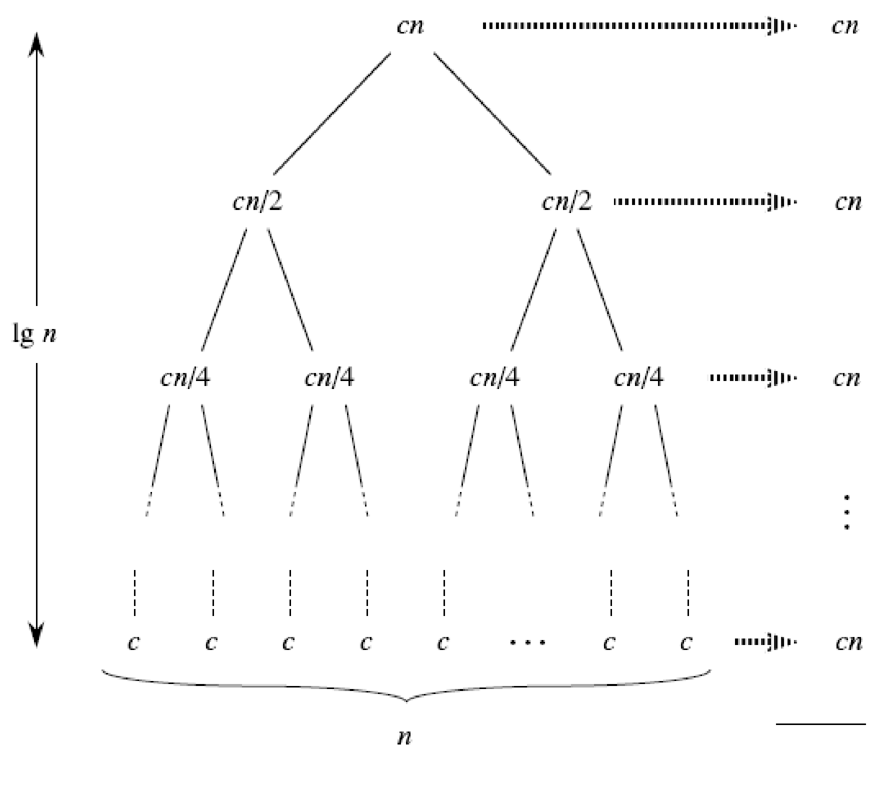
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

To solve this recurrence, let rewrite it to

$$T(n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases}$$

We can view it as a recurrent tree





(d)

The tree has lg n + 1 levels, each level has the cost cn, total is cn lg n + cn

Θ (n lg n)

Homework

What is the worst-case running time of bubble sort?

```
BUBBLESORT(A)
for i ← 1 to length[A]
    do for j ← length[A] downto i + 1
           do if A[j] < A[j − 1]
                  then exchange A[j] ↔ A[j − 1]
```