Lecutre 2 Analysis of Algorithms

By Gou Guanglei

1 Insertion sort

- Insertion sort works the way many people sort a hand of playing cards.

```
INSERTION-SORT(A)
   for j D 2 to A:length
        key = A[j]
        // Insert A[j]  into the sorted sequence A[1..j-1].
        i = j - 1
        while i >0 and A[i]>key
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
```

1.1 Correctness of InsertionSort

- Formally, we will proceed by induction.
  - **Inductive hypothesis**. After iteration i of the outer loop, $A[: i+1]$ is sorted.
  - **Base case**. When $i = 0$, $A[: 1]$ contains only one element, and this is sorted.
  - **Inductive step**.
    - Suppose that the inductive hypothesis holds for $i-1$, so $A[: i]$ is sorted after the $i-1$'st iteration. We want to show that $A[: i+1]$ is sorted after the i'th iteration.
    - Suppose that $j*$ is the largest integer in $0, \ldots, i-1$ so that $A[j*] < A[i]$.
    - Then the effect of the inner loop is to turn

      $[A[0], A[1], \ldots, A[j*], \ldots, A[i-1], A[i]]$

      into

      $[A[0], A[1], \ldots, A[j*], A[i], A[j*+1], \ldots, A[i-1]]$.
    - Therefore, $A[i]$ is **in place**.

      This is because $A[i] > A[j*]$, and by the inductive hypothesis, we have $A[j*] \geq A[j]$ for all $j \leq j*$, and so $A[i]$ **is larger than everything that is positioned before it**. Similarly, by the choice of $j*$ we have $A[i] \leq A[j*+1] \leq A[j]$ for all $j \geq j*+1$, so $A[i]$ **is smaller than everything that comes after it**.
    - Thus, after the i'th iteration completes, $A[: i+1]$ is sorted, and this establishes the inductive hypothesis for i.

1.2 Running time of InsertionSort

- The running time of InsertionSort is about n^2 operations.

- To be a bit more precise, at iteration i, the algorithm may have to look through and move i elements, so that's about $\sum_{i=1}^{n} i = n(n+1)/2$ operations.
- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Usually, we are interested in the **worst-case** running time because
    - It gives an upper bound (because everybody likes a **guarantee**)
    - For some algorithms, the worst case occurs often.
    - Average case is often as bad as the worst case.

2 Kinds of analysis

- **Worst-case: (usually)**
    - $T(n) =$ maximum time of algorithm on any input of size n.
- **Average-case: (sometimes)**
    - $T(n) =$ expected time of algorithm over all inputs of size n.
    - Need assumption of statistical distribution of inputs.
- **Best-case: (bogus)**
    - Cheat with a slow algorithm that works fast on *some* input.

3 Asymptotic analysis

- Ignore machine-dependent constants.
- This type of analysis focuses on the running time of your algorithm as your input size gets very large (i.e. $n \to +\infty$).
- Look at ***growth*** as $n \to +\infty$.

3.1 Asymptotic Notation

1. O-notation upper bound

- $O(g(n)) = \{f(n) : \text{there exist positive constants} c \text{ and } n_0 \text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$
- $g(n)$ is an **asymptotic upper bound** for $f(n)$

  $\lim_{x \to +\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g)$
- $T(n) = f(n) \in O(g(n))$, such that $T(n) \leq cg(n)$. **Notice that $O(g(n))$ is a set of functions.**
- $T(n)$ is proportional to $f(n)$, or better, as n gets large.

example

$$T(n) = 3n^2 + 17$$

$T(n) = O(n^2)$ CORRECT!

$T(n) = O(n^3)$ CORRECT!

2. Ω-notation lower bound

- $\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$
- $g(n)$ is a **lower bound** for $f(n)$

  $\lim_{x \to +\infty} \frac{f(n)}{g(n)} = \infty, \Rightarrow f \in \Omega(g)$
- $T(n) = f(n) \in \Omega(g(n))$, such that $T(n) \geq cg(n)$. **Notice that $\Omega(g(n))$ is a set of functions.**
- $T(n)$ is proportional to $g(n)$, or worse, as n gets large.

example

$T(n) = 3n^2 - 2n$

$T(n) = \Omega(n^2)$ CORRECT!

$T(n) = \Omega(nlogn)$ CORRECT!

3.Θ-notation tight bound

- $\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$
- $g(n)$ is an **asymptotic tight bound** for $f(n)$.

  $\lim_{x \to +\infty} \frac{f(n)}{g(n)} = c, c > 0 \text{ or } c \in \mathbb{R}^+ \Rightarrow f \in \Theta(g)$, $\mathbb{R}^*$ is the set of non-negative real numbers.
- $T(n) \in \Theta(g(n))\ \ if\ \ T(n) \in O(g(n))\ \ and\ \ T(n) \in \Omega(g(n))$.
- $T(n)$ is proportional to $g(n)$ as n gets large.

example

$3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

4. Very informally

- O is like $\leq$
- Ω is like $\geq$
- Θ is like $=$

3.2 Properties

1. Transitivity

- $f(n) = O(g(n))\ \ and\ \ g(n) = O(h(n))\ \ \Rightarrow f(n) = O(h(n))$
- $f(n) = \Theta(g(n))\ \ and\ \ g(n) = \Theta(h(n))\ \ \Rightarrow f(n) = \Theta(h(n))$
- $f(n) = \Omega(g(n))\ \ and\ \ g(n) = \Omega(h(n))\ \ \Rightarrow f(n) = Omega(h(n))$

2. Reflexivity

- $f(n) = O(g(n))$
- $f(n) = \Theta(g(n))$
- $f(n) = \Omega(g(n))$

3. Symmetry

- $f(n) = \Theta(g(n))\ \ if\ \ and\ \ only\ \ if\ \ g(n) = \Theta(f(n))$

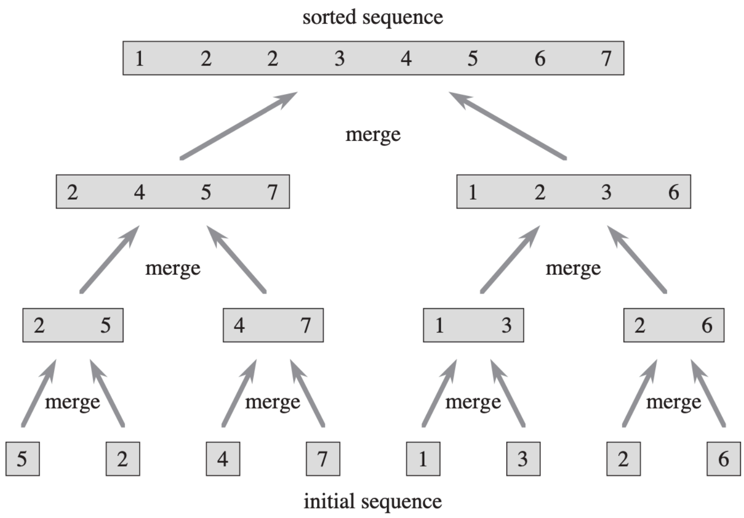
4. Transpose symmetry

- $f(n) = O(g(n))\ \ if\ \ and\ \ only\ \ if\ \ g(n) = \Omega(f(n))$
- $f(n) = o(g(n))\ \ if\ \ and\ \ only\ \ if\ \ g(n) = \omega(f(n))$

4. Recurrences and how to solve them.

4.1 Merge sort

```
MERGE-SORT(A, p, r)
  if p < r
      q = ( p + r ) / 2
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q+1, r)
      MERGE(A, p, q, r)
```



- Correctness of `MERGE−SORT`
  - **Base case.** Suppose that $i = 1$. Then whenever MergeSort returns an array of length 0 or length 1, that array is sorted.
  - **Inductive step.**
    - Suppose `MERGE−SORT` returns a sorted array on intputs of length $\leq i - 1$.
    - Just prove it does for length $\leq i$.
      - $A[p..q]$ and $A[q+1..r]$ are both of length $\leq i - 1$.
      - So by induction, $A[p..q]$ and $A[q+1..r]$ are both sorted.
      - When `Merge` takes as inputs two sorted arrays $A[p..q]$ and $A[q+1..r]$, then it returns a sorted array containing all of the elements of L, along with all of the elements of R.
- The running time of `MERGE−SORT`
  - $T(n) = 2T(n/2) + cn$

4.2 Solving by unrolling

- unroll it to get a summation

example1

$$T(n) = T(n-1) + cn = T(n-2) + c(n-1) + cn = ... = cn + c(n-1) + c(n-2) + ... + c$$

- there are n terms and each one is **at most** cn, $T(n) \leq cn^2$
- the first $n/2$ terms are each **at least** $cn/2$, $T(n) \geq (n/2)(cn/2) = cn^2/4$.
- therefore, $T(n) = \Theta(n^2)$

example2

$$T(n) = n^5 + T(n-1) = n^5 + (n-1)^5 + T(n-2) = n^5 + (n-1)^5 + ... + 1^5$$

- there are n terms each of which is **at most** n^5 , $T(n) \leq n^6$
- the first $n/2$ terms are each **at least** $(n/2)^5$, $T(n) \geq (n/2)(n/2)^5 = (n/4)^6$.
- therefore, $T(n) = \Theta(n^6)$

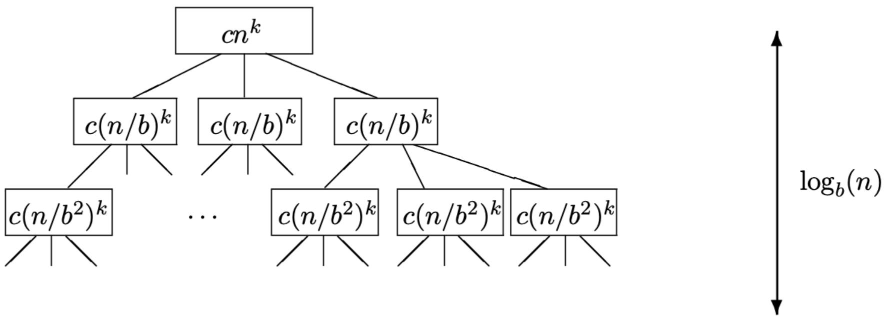
4.3 Solving by substitution method

- Guess the form of the solution.
- Use mathematical induction to find the constants and show that the solution works.

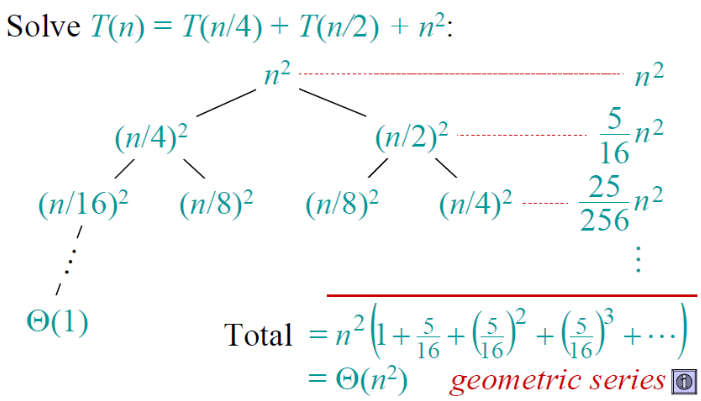
4.4 Recursion tree

- each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.

- we sum *all* the per-level costs to determine the *total cost* of all levels of the recursion.
- Consider the following type of recurrence
  - $T(n) = aT(n/b) + cn^k$
  - $T(1) = c$
    - a means the number of subproblems
    - n/b means the subproblem size
    - cn^k means work dividing and combining



example



Master theorem

- $T(n) = aT(n/b) + cn^k \;\; , \;\; T(1) = c$
  - $T(n) \in \Theta(n^k) \;\; if \;\; a < b^k$
  - $T(n) \in \Theta(n^k logn) \;\; if \;\; a = b^k$
  - $T(n) \in \Theta(n^{log_b a}) \;\; if \;\; a > b^k$
- Proof.
  - To compute the result of the recurrence, we simply need to add up all the values in the tree.
  - The summation is $cn^k[1 + a/b^k + (a/b^k)^2 + (a/b^k)^3 + ... + (a/b^k)^{log_b n}]$
  - Let $r = a/b^k$
  - summation simplifies to $cn^k[1 + r + r^2 + r^3 + ... + r^{log_b n}]$
    - Case 1: $r < 1$. In this case, the sum is a convergent series. $1 + r + r^2 + ... = 1/(1-r)$. So, we can *upper-bound* by $cn^k/(1-r)$ and *lower-bound* by cn^k. Hence, this solves to $\Theta(n^k)$.
    - Case 2: $r = 1$. So the result is $\Theta(cn^k logn)$.
    - Case 3: $r > 1$. In this case, the last term of the summation dominates. We can see this by pulling it out.

      $$cn^k r^{log_b n}[1/r^{log_b n} + r/r^{log_b n} + r^2/r^{log_b n} + r^3/r^{log_b n} + ... + 1]$$

      Since $1/r < 1$, we can now use the same reasoning as in Case 1: the summation is at most

      $1/(1 - 1/r)$ which is a constant. Therefore, we have

$$T(n) \in \Theta(n^k r^{log_b n}) = \Theta(n^k (a/b^k)^{log_b n})$$

As $b^{klog_b n} = n^k$, the formula can be simplified to

$$T(n) \in \Theta(a^{log_b n}).$$

Using $a^{log_b n} = b^{(log_b a)(log_b n)} = n^{log_b a}$, we get

$$T(n) \in \Theta(n^{log_b a}).$$