Part I. Foundations

Chapter 1. The role of algorithms in computing

Algorithm: a well-defined procedure that takes an input and produces an output.

$$\text{Input } (x) \rightarrow [A] \rightarrow \text{output } (y)$$

Example: Algorithm MAX;
Input: List $x = \{a_1, \cdots, a_n\}$;
Body: a series of instructions;
Output: y, the maximum of $a_1, \cdots, a_n$.

An algorithm specifies a *finite process* to compute a function or a relation.

e.g., algorithm MAX computes the following function: $f_{\max}(x) = y$, where $y \geq a, \forall a \in x$.

Questions:

Can an algorithm produce non-unique answers, in other word, different runs of the algorithm produce different results y for the same input x?

What are *deterministic, non-deterministic*, *probabilistic*, and *parallel* algorithms?

- Deterministic algorithms: Given the same input, will always produce the same output.
- Non-deterministic algorithms: May produce different outputs for the same input on different runs.
- Probabilistic algorithms: Make use of randomness or probability in its operation.
- Parallel algorithms: Execute multiple computational tasks in parallel, rather than sequentially.

Computational problems

There are many computational problems in the areas of electrical engineering, biological sciences, manufacturing, internet programming etc.

(1) *search problems*: for which algorithms are required to produce an output y that may be in a complex form.

A search problem corresponds to a general function $f(x) = y$.

(2) *decision problems*: for which algorithms output "yes"/"no".

A decision problem corresponds to a predicate $g(x) = y \in \{0, 1\}$.

Traveling Salesman Problem TSP – search problem
Input: a weighted undirected graph $G = (V, E)$;
Output: a simple cycle containing all vertices in V (Hamiltonian cycle) such that the total cycle weight is the minimum.

A related decision problem:
Input: a weighted undirected graph $G = (V, E)$ and a number k;
Output: "yes" if and only there is a weight at most k Hamiltonian cycle (a cycle that visit each vertex exactly once and returns to the starting vertex) in G.

Claim: The decision problem is not necessarily "easier" than the original search problem.

Actually any "fast" algorithm for the decision problem can be used to quickly construct a minimum Hamiltonian cycle for the search problem. How?

Formally, assume that there is an algorithm A solving the decision problem:

$$\langle G, k \rangle \to [A] \to \text{“}yes'' / \text{“}no''$$

The following process produces a minimum Hamiltonian cycle for any given graph G.

step 1. decide the minimum weight k_0 for G (how?);
step 2. select an arbitrary *unmarked* edge $e = \{u, v\}$;
step 3. let $G' = (V, E - \{e\})$, run A on $\langle G', k_0 \rangle$
step 4. if the answer is “yes”, remove e from G;
otherwise, mark edge e; goto step 2.

Algorithms as a technology to resolve efficiency issues

Efficient use of computer resources such as time and space.

Two situations:

(1) very large input data for “easy” problems;

(2) moderately large input data for “hard” problems.

Chapter 2. Getting started

SORTING PROBLEM

Input: a sequence of n numbers $\langle a_1, \cdots, a_n \rangle$;
Output: a reordering $\langle a'_1, \cdots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Insertion Sort

an iterative process to produce a new list that at each iteration, the list consists of two sublists, a *sorted one* and an *unsorted one*, and the first element in the unsorted list is being inserted into the sorted one.

```
INSERTION-SORT(A)
for j <-- 2 to length[A]
    do key <-- A[j]
    {Insert A[j] into sorted A[1..j-1]}.
    i=j-1
    while i>0 and A[i] >key
       do A[i+1] <-- A[i]
          i=i-1
    A[i+1] <-- key
```

Loop invariant (useful for proving the correctness of algorithms)

at each iteration, the sublist A[1..j-1] consists of the elements originally in the positions [1..j-1] but in sorted order.

properties: initialization, maintenance, and termination

Pseudocode conventions

(1) indentation for block structure;

(2) $\leftarrow$ for assignment, multiple assignments: $x \leftarrow y \leftarrow z$ is same as $y \leftarrow z$ and then $x \leftarrow y$;

(3) only local variables are allowed;

(4) A[i..j] is the subarrray of elements A[i],...A[j];

(5) call-by-value in parameter passing.

Analyzing algorithms

(1) random-access machine (RAM)

(2) primitive operations: add, substract, floor, ceiling, multiply, jump, memory movement, etc. difference: a constant multiplicative factor.

(3) speed between different machines: a constant multiplicative factor.

(4) Turing machine model, the $O(\log n)$ factor.

Analysis of Insertion Sort

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

Assume t_j to be the number of times **while** is executed for every j.

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

$$c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Then for some a, b, c,

$$T(n) \leq a \sum_{j=2}^{n} t_j + bn + c \text{ — } (1)$$

and for some d, e, f,

$$T(n) \geq d \sum_{j=2}^{n} t_j + en + f \text{ — } (2)$$

The best case is when the list is already sorted: $t_j = 1$

The worst case is when the list is reversally sorted: $t_j = j$

So we have to use $t_j = j$. We have

$$T(n) \leq xn^2 + yn + z \text{ for some } x, y, z, \text{ where } x > 0 \text{ — } (3)$$

$$T(n) \geq un^2 + vn + w \text{ for some } u, v, w, \text{ where } u > 0 \text{ — } (4)$$

We will need a simpler notation for $T(n)$.

size of the input: n, the number of bits used to encode the input.

For some problems, we may use different definitions of the input size.

running time of an algorithm: $t(n)$, the number of primitive operations executed, defined as a function in the input size n.

worst-case running time: the upper bound on running time for any input.

average-case running time: the running time "on average" or running time on a randomly chosen input assuming all inputs of a given size (n) are equally likely.

order of growth: e.g., $an^2 + bn + c$, the growth rate depends on an^2 as n grows if $a > 0$.

Designing algorithms

Divide-and-conquer approach: appropriate for problems that can be solved recursively.

(1) **divide** the problem into a number of subproblems.

(2) **conquer** the subproblems by solving them recursively or in a straightforward manner if the size of a subproblem is small enough.

(3) **combine** the solutions to the subproblems into a solution for the original problem.

There are some divide-and-conquer approaches for *Sorting Problem*.

e.g., "splitting a list into two of equal size" leads to Merge-Sort algorithm

```
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)
```

Analysis of Merge-Sort

$n = r - p + 1$, assume that n is a power of 2.

(1) time for divide: c_1 for split the list into two sublists;

(2) time for conquer: $2T(n/2)$ for recursively solve subproblems

(3) time for combine: $c_2 n$ for merging two length $n/2$ sorted sublists;

Recurrence:

$$T(n) = 2T(n/2) + c_2 n + c_1 \quad \text{when } n > 1$$

$$T(n) = 0 \quad \text{when } n = 1$$

How to solve the recurrence?

$T(n) = 2T(n/2) + c_2 n + c_1$

$2T(n/2) = 2^2 T(n/2^2) + 2c_2 n/2 + 2c_1$

$2^2 T(n/2^2) = 2^3 T(n/2^3) + 2^2 c_2 n/2^2 + 2^2 c_1$

. . .

$2^k T(n/2^k) = 2^{k+1} T(n/2^{k+1}) + 2^k c_2 n/2^k + 2^k c_1$

Let $n/2^{k+1} = 1$, then $k + 1 = \log_2 n$

Then $T(n) = 2^{k+1} T(1) + (k + 1)\, c_2 n + c_1 \sum_{i=0}^{k} 2^i$

$T(n) = 0 + c_2 n \log_2 n + c_1(2^{k+1} - 1) = c_2 n \log_2 n + c_1(n - 1)$

How fast does $T(n)$ grow?

When n is big enough, there is a constant $a > 0$ such that

$T(n) \leq an \log_2 n$.

$T(n)$ cannot grow faster than $an \log_2 n$ for some constant $a > 0$.

Apparently, there is a constant $b > 0$ such that

$T(n) \geq bn \log_2 n$

$T(n)$ grows faster than $bn \log_2 n$ when n is large enough.

Chapter 3. Growth of Functions

<u>Asymptotic notations:</u>

$O(g(n))$ = { $f(n)$: $\exists c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$, for all $n \ge n_0$ }

$\Omega(g(n))$ = { $f(n)$: $\exists c > 0$, $n_0 > 0$ such that $0 \le cg(n) \le f(n)$, for all $n \ge n_0$ }

$\Theta(g(n))$ = { $f(n)$: $\exists c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that
$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$, for all $n \ge n_0$ }

$o(g(n))$ = { $f(n)$: for $\forall c > 0$, $\exists n_0 > 0$ such that $0 \le f(n) < cg(n)$,
for all $n \ge n_0$ }

$\omega(g(n))$ = { $f(n)$: for $\forall c > 0$, $\exists n_0 > 0$ such that $0 \le cg(n) < f(n)$,
for all $n \ge n_0$ }

other notations and functions

floors and ceilings

modular arthmetic

polynomials

exponentials

logarithms

Stirling's approximation $n = \sqrt{2\pi n}(n/e)^n(1+\Theta(1/n))$

Fibonacci numbers: 0 1 1 2 3 5 8 13........

Chapter 4. Recurrences

Techniques to solve recurrences

(1) substitution method – guess and use of math induction

example: $T(n) = 3/2T(2n/3) + n$

($T(n) = d$ for $n = 1$)

guess: $T(n) \leq cn \log n + d$

verify:

(1) base case $n = 1$: $T(1) = d \leq 0 + d$

(2) general case: $T(n) = 3/2T(2n/3) + n$

$3/2[c(2n/3) \log(2n/3) + d] + n$

$= cn (\log n + \log 2/3) + 3/2d + n$

$= cn \log n + d - cn \log 3/2 + 1/2d + n$

$\leq cn \log n + d$ when

$1/2d + n \leq cn \log 3/2$, i.e., $c \geq \frac{1/2d+n}{n \log 3/2}$

(2) changing variables

example: $T(n) = 2T(\sqrt{n}) + \log_2 n$

define $m = \log_2 n$, i.e., $n = 2^m$

then $T(2^m) = 2T(2^{m/2}) + m$

rename the function: $S(m) = T(2^m)$

$S(m) = 2S(m/2) + m$

solve it, we have $S(m) = O(m \log m)$

so $T(n) = T(2^m) = O(m \log m) = O(\log n \log \log n)$.

(3) Recursive-tree method

also based on *unfolding* the recurrence to make a recursive-tree.

(1) $T(n)$ is a tree with non-recursive terms as the root and recursive terms as its children.

(2) for each child, replace it with then non-recursive terms and producing children that are then recursive terms

(3) repeat (2), expand the tree until all children are the base case.

example $T(n) = 3T(n/4) + cn^2$

Chapter 5. Probabilistic analysis and randomized algorithms

(1) probabilistic analysis of algorithms

(2) randomized algorithms

```
HIRE-ASSISTANT (n)

1. best <- 0 {candidate 0 is a dummy candidate}
2. for i <- 1 to n
3   do interview candidate i
4.    if candidate i is better than candidate best
5.      then best <- i
6            hire candidate i
```

analyzing the spending in this hiring process.

worst-case n x cost of hiring

average case?

Probabilistic analysis

indicator random variable X_A associated with event A. Then

$X_A = 1$ if A occurs;
$X_A = 0$ if A does not occur.

Let X_i be the indicator random variable associated with the event that the ith candidate is hired.

Then $X = X_1 + \cdots + X_n$, random variable indicating the number of times we hire a new assistant.

average times $= \Sigma_{k=1}^{n} k Prob(X = k)$ – called expected number of X, denoted as $E[X]$.

$E[X] = E[\Sigma_{i=1}^{n} X_i] = \Sigma_{i=1}^{n} E[X_i]$

$E[X_i] = 1/i$ assuming candidates arrive at random order.

$E[X] = \Sigma_{i=1}^{n} 1/i = \ln n + O(1)$

Randomized algorithms

imposing a distribution on *any given input*, e.g., randomly permute candidates. randomization is in algorithms, not in the input distribution.

Each run of a randomized algorithm may produce a different result.

```
RANDOMIZED-HIRE-ASSISTANT(n)

1. randomily permute the list of candidates
2. best <- 0
3. ...
```

average hiring cost remains the same.