2 Analysis of Algorithms

2.1 An overview

Reading: CLRS 1.1 and 1.2

- An algorithm is any well-defined computational procedure that takes some value or set of values as input and produces some value or set of values as output.
- Design of algorithms: Design techniques such as divide-and-conquer, greedy method, and dynamic programming plus use of appropriate data structures
- Description of algorithms: Shows the ideas of algorithms in a clear and clean way.
- Proof of correctness of algorithms: Requires special proof techniques and can be tricky sometimes.
- Analysis of the time complexity of algorithms: Must be independent of (1) makes/models of computers, (2) programming languages, (3) input data, and (4) skills of individual programmers.

  Our solution is

  1. Use any abstract computing model such as Turing Machine or its high-level equivalence Random Access Machine and count number of steps of the algorithm within the chosen model to measure time. Note all such computing models are polynomially-related so it really does not matter which model to choose.
  2. Use pseudocode to describe algorithms, which is the combination of natural language, math language, and programming language. Remember the description of an algorithm should be concise but also easy to follow.
  3. Represent time complexity (or number of basic steps) of an algorithm as a function of input size (not value).
  4. Express the time complexity function in asymptotic notation. This shows the performance of an algorithm for large-size input and makes it easy to compare the time complexity of different algorithms and to classify problems according to the time complexity of their best algorithms.
- *Example:* The evolution of algorithms for Linear Programming (LP).

2.2 Worst-case analysis

Reading: CLRS 2.1 and 2.2

- Definition:

  I_n: Any instance of size n;

  $t(I_n)$: Time (# of basic steps) spent on I_n by the algorithm;

  $T(n)$: Worst-case time complexity of any instance of size n, $T(n) = \max_{\forall I_n}\{t(I_n)\}$.
- Insertion sort: $A \Rightarrow A$ sorted in increasing order.

  Insertion_Sort(A)

  for $j = 2$ to $A.length$

  $key = A[j]$

  //Insert $A[j]$ into sorted $A[1..j-1]$

  $i = j-1$

  while $i > 0$ and $A[i] > key$

  $A[i+1] = A[i]$

  $i = i-1$

  $A[i+1] = key$

  To insert $A[j]$ into the sorted $A[1\ldots j-1]$, the algorithm will make at most $j-1$ comparisons and $j-1$ shifts. So the overall time complexity, which is dominated by the number of comparisons and the number of shifts, is at most $2\sum_{j=2}^{n}(j-1) = \Theta(n^2)$, where $n = A.length$.

- Euclid's algorithm (300 B.C.): Greatest common divisor.

  Recursively defined: If $n \neq 0$, then $gcd(m,n) = gcd(n, m \bmod n)$; If $n = 0$, then $gcd(m,n) = m$.

  Euclid(m,n) //Assume $m \geq n$

  while $n > 0$

  $t = m \bmod n$

  $m = n$

  $n = t$

  return m

  The time complexity of the algorithm is determined by the number of iterations of the while-loop. Let it be k. Also let m_i and n_i be the values of m and n at the end of the ith iteration, respectively.

  We observe that (1) $n_k = 0$, $n_i \geq 1$ for $i = 0, \ldots, k-1$, and $n_0 > \cdots > n_k$; (2) $m_i = n_{i-1} > n_i$, for $i = 1, \ldots, k$; (3) $n_i = m_{i-1} \bmod n_{i-1} < \frac{m_{i-1}}{2} = \frac{n_{i-2}}{2}$ for $i = 2, \ldots, k$. (Note: If $a \geq b$, then $a \bmod b < \frac{a}{2}$.)

  We then have $1 \leq n_{k-1} < \frac{n_{k-3}}{2} < \frac{n_{k-5}}{2^2} < \cdots < \frac{(n_1,n_0)}{2^{\lceil k/2 \rceil - 1}} \leq \frac{n_0}{2^{\lceil k/2 \rceil - 1}}$.

  Thus, $\frac{k}{2} - 1 \leq \lceil \frac{k}{2} \rceil - 1 < \log n_0$, which implies $k \leq 2 + 2\log n_0$.

  So $T(m,n) = O(k) = O(\log n)$.

- Multiplying two integers x and y:

  Assume that x (multiplicand) and y (multiplier) have the same number of figures, $n = 2^k$, denoted $(x)_n$ and $(y)_n$. Then $(x)_n = 10^{\frac{n}{2}} \cdot (a)_{\frac{n}{2}} + (b)_{\frac{n}{2}}$ and $(y)_n = 10^{\frac{n}{2}} \cdot (c)_{\frac{n}{2}} + (d)_{\frac{n}{2}}$.

  So $(x)_n \cdot (y)_n = 10^n((a)_{\frac{n}{2}} \cdot (c)_{\frac{n}{2}}) + 10^{\frac{n}{2}}((a)_{\frac{n}{2}} \cdot (d)_{\frac{n}{2}} + (b)_{\frac{n}{2}} \cdot (c)_{\frac{n}{2}}) + (b)_{\frac{n}{2}} \cdot (d)_{\frac{n}{2}}$. The problem of multiplying two integers of size n is now reduced to four subproblems of multiplying two integers of size $\frac{n}{2}$.

  $T(n) = 4T(\frac{n}{2}) + n = \Theta(n^2)$ by the master theorem.

  Now let $s = (a)_{\frac{n}{2}} \cdot (c)_{\frac{n}{2}}$, $t = (b)_{\frac{n}{2}} \cdot (d)_{\frac{n}{2}}$, and $r = ((a)_{\frac{n}{2}} + (b)_{\frac{n}{2}}) \cdot ((c)_{\frac{n}{2}} + (d)_{\frac{n}{2}})$. Then $(x)_n \cdot (y)_n = 10^n s + 10^{\frac{n}{2}}(r - s - t) + t$. Only three subproblems have to be solved in the recursion.

  $T(n) = 3T(\frac{n}{2}) + n = \Theta(n^{\log_2 3}) = \Theta(n^{1.585})$ by the master theorem.

2.3 Average-case analysis

Reading: CLRS 2.2

- Definition:

  I_n: Any instance of size n;

  $t(I_n)$: Time (# of basic steps) spent on I_n by the algorithm;

  $p(I_n)$: Probability that I_n appears as input;

  $T(n)$: Average-case time complexity of any instance of size n, $T(n) = \sum_{\forall I_n} p(I_n)t(I_n)$.

- Insertion sort:

  Assumptions: (1) Distinct items; (2) Each permutation with equal probability to occur.

  $T(n) =$ average # of pairwise comparisons $= \sum_{j=2}^{n} c_j$, where c_j is the average # of comparisons to insert $A[j]$ (*key*) into the sorted $A[1 \ldots j-1]$. For *key*, there are j possible positions, with probability $\frac{1}{j}$ for each.

  So $c_j = \frac{1}{j} \cdot 1 + \frac{1}{j} \cdot 2 + \cdots + \frac{1}{j} \cdot (j-1) + \frac{1}{j} \cdot (j-1) = \frac{1}{2}(j+1) - \frac{1}{j}$.

  Therefore, $T(n) = \sum_{j=2}^{n}(\frac{1}{2}(j+1) - \frac{1}{j}) = \frac{1}{4}n^2 + \frac{3}{4}n - H_n = \Theta(n^2)$.

- Binary search

  Given $A[1 \ldots n]$ sorted and x (a query). Is x a member of A?

  Assumptions: (1) $x \in A$; (2) Distinct items in A; (3) $Pr(x = A[i]) = \frac{1}{n}$ for any $i = 1, 2, \ldots, n$; (4) $n = 2^k - 1$ for some k.

  Binary search can be illustrated by a decision tree of n nodes (full binary tree with k levels).

  Example: What is the decision tree for $n = 7$?

  Average-case time is then the average length of a path from the root to a tree node.

Level	# of nodes	amount to add to average length
1	1	$\frac{1}{n} \cdot 1$
2	2	$\frac{1}{n} \cdot 2 + \frac{1}{n} \cdot 2$
3	4	$\frac{1}{n} \cdot 3 + \frac{1}{n} \cdot 3 + \frac{1}{n} \cdot 3 + \frac{1}{n} \cdot 3$
$\cdots$	$\cdots$	$\cdots$
k	2^{k-1}	$\frac{2^{k-1}}{n} \cdot k$

$$T(n) = \sum_{i=1}^{k} \frac{2^{i-1}}{n} \cdot i = \frac{1}{n} \sum_{i=1}^{k} i \cdot 2^{i-1} = \frac{1}{n}(k \cdot 2^k - 2^k + 1) = k + \frac{k}{n} - 1 = \Theta(\log n).$$

- Binary search trees (BST)

  What is the average number of comparisons, $T(n)$, needed to insert n distinct random elements into an initially empty BST? First, $T(0) = T(1) = 0$.

  Assume that $A = (a_1, \ldots, a_n)$ is the list given and that $B = (b_1, \ldots, b_n)$ is A sorted increasingly. That A is a random sequence implies that a_1 is equally likely to be b_j for any $1 \leq j \leq n$. Consider the tree obtained after the insertion of all n numbers, which has $a_1(b_j)$ as the root, $b_1, \ldots, b_{j-1}$ in the left subtree, and $b_{j+1}, \ldots, b_n$ in the right subtree.

  So $T(n) = \frac{1}{n}\sum_{j=1}^{n}(n-1+T(j-1)+T(n-j)) = n-1+\frac{2}{n}\sum_{j=0}^{n-1} T(j)$ for $n \geq 2$.

  To solve this recurrence, we guess $T(n) = O(n \log n)$, i.e., $T(n) \leq cn \ln n$ for some c. We prove by induction that this is indeed the case.

2.4 Amortized analysis

Reading: CLRS 17.1–17.3

- Amortization:

  (1) To put money aside at intervals, as a sinking fund, for gradual payment of a debt;

  (2) To average the time (cost) required to perform a sequence of data structure operations over all operations performed.

- Motivation:

  A sequence of n data structure related operations, rather than just a single operation, is performed. (An operation may change the data structure, thus affect the next operation.) What is the total time complexity of the entire sequence?

  Worst-case analysis: Sum of worst-case time of each operation (which may never be achieved, thus not tight and overly pessimistic).

  Average-case analysis: Averaging over all possible inputs and involving probability.

  Amortized analysis: Averaging over a worst-case sequence to determine an amortized cost for each operation, which can be the total cost of the sequence divided by the number of operations, i.e., $T(n)/n$, or any costs as long as they add up to the total cost $T(n)$. Amortized analysis often gives a tight worst-case time bound.

- Three techniques:

  Aggregate analysis: Attempt to consider the entire sequence as a whole and show for all n, a sequence of n operations takes worst-case time $T(n)$ in total, therefore, the amortized cost per operation is $T(n)/n$.

  Accounting method (banker's view): Represent prepaid work as credit stored with specific objects within the data structure.

  Potential method (physicist's view): Represent prepaid work as potential energy that can be released for future operations. The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

- Example: Stack manipulation

  Three types operations: $push(S,x)$, $pop(S)$ and $multipop(S,k)$.

  Assume a sequence contains n operations defined above and initially the stack is empty.

  Question: What is the total time/cost of a sequence of n operations?

- Worst-case analysis: $O(n^2)$
- Aggregate analysis: $2n = O(n)$. (At most n items pushed to the stack and each may result in a a later pop.)
- Accounting method: Coin-operated computer: 1 credit $\Rightarrow$ 1 time unit $\Rightarrow$ 1 push/pop.
  Sequence of operations: $O_1, \ldots, O_n$, where operation O_i is allocated with credits c_i.
  Assumptions: (1) Unused credits are carried over to later operations; (2) Operations can borrow credits as long as any debt is paid off eventually.
  If all sequences of length n can be performed with the allocated credits, the total time is no larger than $\sum_{i=1}^{m} c_i$.
  Keys: (1) How to pick the smallest possible c_i; (2) How to prove all sequences can be performed with the allocated credits $c_1, \ldots, c_n$.
  For the stack manipulation example, let $c_i = 2$ for all *push* and $c_i = 0$ for *pop* and *multipop*. For any *push*, one credit is used to execute the operation itself, while the other is saved for pop (of the same element) in a later *pop* or *multipop* operation. So in other words, each *push* is paid right away and each *pop* or *multipop* is paid for by saved credit/credits. So the total time is bounded from above by $\sum_{i=1}^{n} c_i \leq 2n = O(n)$.
- Potential method: Define a potential function $\Phi : D \to R$. Assume that an operation takes t_i time units to change the data structure from D_{i-1} to D_i. Then the amortized time of the operation is

  $$a_i = t_i + \Phi(D_i) - \Phi(D_{i-1}),$$

  and the total time for the sequence is

  $$\sum_{i=1}^{n} t_i = \sum_{i=1}^{n} (a_i - \Phi(D_i) + \Phi(D_{i-1})) = \Phi(D_0) - \Phi(D_n) + \sum_{i=1}^{n} a_i \leq \sum_{i=1}^{n} a_i \quad \text{if } \Phi(D_0) \leq \Phi(D_n).$$

  Key: How to choose Φ.
  Consider the stack manipulation example. The underlining data structure is the stack. Define $\Phi(D) =$# of items in the stack. Since $\Phi(D_0) = 0$ and $\Phi(D_n) \geq 0$, then $\Phi(D_0) \leq \Phi(D_m)$.
  If O_i is a *push*, then $a_i = t_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.
  If O_i is a *pop*, then $a_i = t_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (-1) = 0$
  if O_i is a *multipop*, then $a_i = t_i + \Phi(D_i) - \Phi(D_{i-1}) = k + (-k) = 0$.
  Therefore, the total time is $\sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i \leq 2n = O(n)$.

- Example: Binary counter

  A k-bit binary counter is implemented by an array $A[0 \ldots k-1]$, where $A.length = k$. A binary number x stored in the counter can be defined as $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially $x = 0$, thus $A[i] = 0$ for $i = 0, 1, \ldots, k-1$. The operation of interest is to increment the value in the counter. We have the following procedure to implement the operation INCREMENT:

  INCREMENT(A)
  $i = 0$
  while $i < A.length$ and $A[i] == 1$
  $A[i] = 0$
  $i = i + 1$
  if $i < A.length$
  $A[i] = 1$

  Starting from $x = 0$, what is the total time of performing a sequence of n INCREMENT operations?

  - Worst-case analysis: $O(kn)$
  - Aggregate analysis: Consider a 4-bit counter as an example. Let $n = 8$.

x	$A[3]$	$A[2]$	$A[1]$	$A[0]$	total cost
0	0	0	0	0	0
1	0	0	0	1	$0+1=1$
2	0	0	1	0	$1+2=3$
3	0	0	1	1	$3+1=4$
4	0	1	0	0	$4+3=7$
5	0	1	0	1	$7+1=8$
6	0	1	1	0	$8+2=10$
7	0	1	1	1	$10+1=11$
8	1	0	0	0	$11+4=15$

Observe that the total cost is also the number of bit-flips. The worst-case analysis assumes that in performing an INCREMENT operation each of the k bits is being flipped once. However, from the above example, bit $A[0]$ has been flipped n times in all n operations, bit $A[1]$ has been flipped $n/2$ times, bit $A[2]$ has been flipped $n/2^2$ times, etc.. So the total time is $\sum_{i=0}^{k-1} n/2^i < 2n = O(n)$.

– Accounting method: 1 credit $\Rightarrow$ 1 time unit $\Rightarrow$ 1 bit flip.

Note that to perform an INCREMENT, there is only one bit that is flipped from 0 to 1 but zero or more bit that are flipped from 1 to 0. We then assign 2 credits to each operation in the sequence, in which one credit is used to flip a 0 to 1 and another credit is saved to flip a 1 to 0 in a later operation. Therefore, the total time cost is no more than $2n$, or $O(n)$.

– Potential method: The underlining data structure D is the k-bit counter (array). Define the potential $\Phi(D) = \#$ of 1's in the counter. Since $\Phi(D_0) = 0$ and $\Phi(D_n) \geq 0$, then $\Phi(D_0) \leq \Phi(D_n)$.

For any INCREMENT operation in the sequence, say I_i, WLOG, assume that I_i will result in f_i flips from 1 to 0 and one or zero (for the case when there are all 1's in the counter) flip from 0 to 1. Then the actual cost of I_i is $t_i \leq f_i + 1$. It is also easy to verify that the increase of the potential caused by I_i is $\Phi(D_i) - \Phi(D_{i-1}) \leq 1 - f_i$. So the amortized cost of I_i is

$$a_i = t_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (f_i + 1) + (1 - f_j) = 2.$$

Therefore, the total time is $\sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i \leq 2n = O(n)$.

2.5 Probabilistic analysis

Reading: CLRS 5.1–5.3

- Overview: When applying the concept of randomization to algorithms, there are usually two views. One is to consider the input to be structured random or following some probabilistic distribution while the algorithms are deterministic without randomness. This results in the average-case analysis. The other is to make no assumption about the input but to allow algorithms to behave random with the use of random process (such as the random number generator). This results in the probabilistic analysis.

- Some simple examples of randomized algorithms: randomized linear search and randomized quick sort.

- Example: Contention resolution

  There are n processes $P_1, P_2, \ldots, P_n$, each competing for access to a single shared database. Imagine time as being divided into discrete rounds. The database can be accessed by at most one process in a single time round; if more than one processes attempt to access it simultaneously, all processes are locked out for the duration of that round. Assuming no communication exists among processes, what is a good protocol with which the processes can access the database on a somewhat regular and equitable basis?

  A randomized algorithm to smooth out contention: For some carefully selected $0 < p \leq 1$, each process will attempt to access the database in each round with probability p, regardless of the decisions of other precesses.

  Probabilistic analysis: What is the probability for a process to succeed in accessing the database in a round?

  – For any process P_i and time round t, define $A[i,t]$ to be the event that P_i attempts to access the database in round t. Clearly, $Pr[A[i,t]] = p$ and $Pr[\overline{A[i,t]}] = 1 - p$.

  – Define $S[i,t]$ to be the event that process P_i succeeds in accessing the database in round t. So $S[i,t] = A[i,t] \wedge (\wedge_{j \neq i} \overline{A[j,t]})$ and $Pr[S[i,t]] = Pr[A[i,t]] \cdot \Pi_{j \neq i} Pr[\overline{A[j,t]}] = p(1-p)^{n-1}$. The maximum success probability $Pr[S[i,t]]$ is achieved by setting $p = 1/n$, yielding $Pr[S[i,t]] = (1/n)(1-(1/n))^{n-1}$.

  – A helpful result from basic calculus: (a) The function $(1-(1/n))^n$ converges monotonically from $1/4$ up to $1/e$ as n increases from 2; (b) The function $(1-(1/n))^{n-1}$ converges monotonically from $1/2$ down to $1/e$ as n increases from 2.

  – From the above result, we get $1/(en) \leq Pr[S[i,t]] \leq 1/(2n)$, and hence $Pr[S[i,t]]$ is asymptotically equal to $\Theta(1/n)$.

  Probabilistic analysis: What is the probability that a process has not yet succeeded after a certain number of rounds?

  – Define $F[i,t]$ to be the event that process P_i does not succeed in any of the rounds 1 through t. Then $Pr[F[i,t]] = Pr[\wedge_{r=1}^{t} \overline{S[i,r]}] = \Pi_{r=1}^{t} Pr[\overline{S[i,r]}] = [1-(1/n)(1-(1/n))^{n-1}]^t \leq (1-1/(en))^t$.

- If we set $t = \lceil en \rceil$, we get $Pr[F[i,t]] \leq (1-1/(en))^{\lceil en \rceil} \leq (1-1/(en)^{en} \leq 1/e$.
- If we set $t = \lceil en \rceil \cdot (c \ln n)$, then $Pr[F[i,t]] \leq (1-1/(en))^t = ((1-1/(en))^{\lceil en \rceil})^{clnn} \leq e^{-c \ln n} = n^{-c}$.
- Conclusion: Asymptotically, after $\Theta(n)$ rounds, the probability that P_i has not yet succeeded is bounded by a constant; and between then and round $\Theta(n \ln n)$, this probability drops to a small quantity, bounded by an inverse polynomial in n.

Probabilistic analysis: What is the probability that all processes succeed at least once in a given number of rounds?

- Define F_t be the event that some process has not yet succeeded in accessing the database after t rounds. Then $F_t = \vee_{i=1}^n F[i,t]$.
- The Union Bound: Given events $E_1, E_2, \ldots, E_n$, $Pr[\vee_{i=1}^n E_i] \leq \sum_{i=1}^n Pr[E_i]$. Or in words, the probability of the union of events is uppper-bounded by the sums of their individual probabilities.
- $Pr[F_t] \leq \sum_{i=1}^n Pr[F[i,t]]$. To make the sum small thus tight for the bound, $Pr[F[i,t]]$ for each i needed to be significantly smaller than $1/n$. Choosing $t = \Theta(n)$ will not be good enough. But if we choose $t = \lceil en \rceil \cdot (c \ln n)$, we will have $Pr[F[i,t]] \leq n^{-c}$ for each i, which is what we want. Specifically, let $t = 2\lceil en \rceil \cdot \ln n$, we have $Pr[F_t] \leq \sum i = 1^n Pr[F[i,t]] \leq n \cdot n^{-2} = 1/n$.
- Conclusion: With probability at least $1-(1/n)$, all processes succeed in accessing the database at least once within $t = 2\lceil en \rceil \cdot \ln n$ rounds.