February 25th, 2014 Practice Questions and Reading Material

Introduction to Concepts of Algorithmic Design, Computing Running Times

Reading material

- Chapters 0.1, 0.2, 0.3 from DPV
- Chapters 1.2, 3.1, 3.2 from CLRS2

Practice questions

- **You may be provided with an algorithm for computing the nth Fibonacci number and then be asked to compute its running time (think in terms of it complexity). Alternatively, you may be asked to provide an efficient algorithm for computing Fibonacci numbers.**

Base case

Recursive step

```
function fib1(n)
if n = 0: return 0
if n = 1: return 1
return fib1(n - 1) + fib1(n - 2)
```

"Caching"

```
(n) function fib2
if n=0 return 0
create an array f[0 . . . n] f[0] =0,f[1] =1
for i = 2 . . . n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]
```

- **Give a justification why imporovement in hardware are typically not sufficient to make an algorithm with exponential running time reasonably tractable.**

> But technology is rapidly improving—computer speeds have been doubling roughly every 18 months, a phenomenon sometimes called Moore's law. With this extraordinary growth, perhaps fib1 will run a lot faster on next year's machines. Let's see—the running time of fib1(n) is proportional to $2^{0.694n} \approx (1.6)^n$, so it takes 1.6 times longer to compute $F_{n+1} than F_n$. And under Moore's law, computers get roughly 1.6 times faster each year. So if we can reasonably compute F_{100} with this year's technology, then next year we will manage F_{101}. And the year after, F_{102}. And so on: just one more Fibonacci number every year! Such is the curse of exponential time.

Take a exponential algo, say naive fibonacci. If you want to calculate F_{100}, that's 100^2 computational steps. Computing power double every 18 months, and notice that "doubling" (2n) is "slower than" exponential (n^2). This means w/out better algo, one more fibonacci every year.

- **We have 3 algorithms for the same problem where the first runs in exponential time, the second in logarithmic, and the thir in linear time as a function of the same input. Which algorith do you prefer to use?**

quadratic n^2

Name	Running time
Exponential	$O(a^n)$
Logarithmic	$(\log(n))$
Linear	$O(n)$

- **You may be provided running times of different algorithms and asked to show which running time dominates asymptotically.**

if there is a constant such that $f(x) \leq C+g(x)$

- **Provide the definition of *O*-notation (or Theta or Omega).**

$f = O(g)$ is "f grows no faster than g",
$f = \Omega(g)$ is "f grows no slower than g"
$f = \Theta(g)$ is "f grow like g"

  Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals. We say f = O(g) (which means that "f grows no faster than g") if there is a constant $c > 0$ such that $f(n) \leq c \times g(n)$.

  - Just as $O(\cdot)$ is an analog of $\leq$, you can define the other analyses as such:
    - * $f = \Omega(g)$ means $g = O(f)$.
    - * $f = \Theta(g)$ means $f = O(g)$ and $f = \Omega(g)$.

- **You may be provided certain statements about asymptotic analysis of running times and asked to show if they are correct or not. For instance, "n to the a dominates n to the b if a is greater than b.**

Related exercises

- DPV: 0.1, 0.2, 0.3
- CLRS2: 3.1-1, 3.1-2, 3.1-3, 3.1-4, 3.2-4
- CLRS2: 1.1, 3-2, 3-3, 3-6

Number Theoretic Algorithms Complexity of adding and multiplying 2 n-bit numbers, modulo arithmetic

Reading material

- Chapters 1.1,1.2 from DPV

CLRS2 31.2 Greatest common divisor

- In this section, Euclid's algorithm for computing the GCD of two integers efficiently.
  - Suprising connection with Fibonacci numbers.
    - This yields the worst case.
- Restricted to nonegative integers.

$$\gcd(a, b) = \gcd(|a|, |b|)$$

- One way of characterizing the problem:

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$
$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r}$$
$$\gcd(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)}$$

The GCD is going to be the product of the minimum prime factors of a, b

- Proof
  - We first show that the GCD of a and b divide each other.
    - If $d = \gcd(a, b)$, then $d|a$ and $d|b$.
  - Because $a \bmod b$ is a linear combination of a and b, $d|(a \bmod b)$.
  - Because $d|b$ and $d|(a \bmod b)$, $d|\gcd(b, a \bmod b)$
  - Equivilently,

$$\gcd(a, b) | \gcd(b, a \bmod b)$$

  - You can show that the reversed is true to, and because if you can invert the divisible operator, the two things are plus or minus equal

primality: is N prime?

factoring: express N as the product of prime factors.

Benefit?: RSA

Practice questions

- **What is the "primality" problem and what is the "factoring" problem? Which one of the two is tractable? What are the advantages of the intractability of the other problem?**

  Factoring problem Given a number N, express it as a product of prime factors.

  Computationally intractable, fastest is exponential. This makes RSA work because keys and factoring large multiples.

  Primality problem Given a number N, determine whether it is a prime.

  Computationally tractable.

- **How many digits do you need to represent a number N in base b?**
  - With k digits in base b we can express numbers up to $b^k - 1$.
    - * For instance, in decimal, three digits get us all the way up to $999 = 10^3 - 1$.
  - By solving for k, we find that $\lceil \log_b(N+1) \rceil$ are need to write N in base b.

you can express $b^K - 1$ numbers with base b and digits k. Solve for k.

$K = \lceil \log(N+1) \rceil$

- **How much does the size of the representation of a number changes when we change bases?**
  - Recall the rule for converting logarithms from base a to base b:

$$\log_b N = (log_a N)/(log_a b)$$

    - * So the size of integer N in base a is the same as its size in base b, times a constant factor $log_a b$.
    - * In big-O, the base is irrelevant, and we write the size simply as

$$O(\log N)$$

converting bases involves a division by another log. Base doesn't matter to big-Oh. $\log N$

- **What is the running time of the addition operation of a number for two n-bit numbers (think in terms of bit complexity)?**
  - Suppose x and y are n bits long.
  - The sume of x and y is *at most* $n + 1$ bits long.
  - Each bit of the sume is computed in a fixed amount of time.
  - The total running time for addition will be

$$c_0 + c_1 n$$

In other words, *linear*.

$$O(n)$$

You have to at least look at every bit of both numbers, and sum is done in fixed time $O(n)$

- **What is the running time of the tradition multiplication operation taught in grade school for two n-bit numbers (think in terms of n-bit complexit)?**
  - To compute each row, either "X" or "0", left-shifted
  - The rows are in the order of $2n$.
  - You have to sum them up, and you do this pairwise.
  - If you do n times an operation which costs n, your running time is going to be n^2.

You have to create rows in the order of $2n$, then sum them pairwise. There are n operations that cost n

$O(n^2)$

- **Provide a recursive formula for the multiplication of two numbers.**

```
function multiplication(x, y) {
    if (y == 1) {
        return x;
    }

    else {
        return x + multiplication(x, y - 1);
    }
}
```

base case! decrementing y

Recursive step.

- **What is the complexity of modular addition and modular multiplication for two n bit numbers?**
  - **Addition**: $O(n)$
  - **Multiplication**: $O(n^2)$

Related exercises

- DPV: 1.1, 1.2, 1.3, 1.4, 1.6, 1.7, 1.8, 1.9, 1.10, 1.31

RSA Cryptosystem, Fermat's Little Theorem and Modular Exponentiation

Reading material

- Chapters 1.2,1.4 from DPV
- Chapter 31.3, 31.6, 31.7 from CLRS2

Practice questions

- **Give a private key protocol for a cryptography application. Given an example it can be compromised.**
  - There is the classic "codebook" private-key scheme, where Alice and Bob meet before hand and agree on a secret code.
  - The way that this is compromised is if Eve gets the codebook or knows a given message and backtraces the code from it.

Private key protocols require meeting to agree on a key. Consider a Cipher. Eve must crack the code w/ the book or a known message.

  - ~~Alternatively, the answer you're looking for might be that you can encode and decode with public keys like:~~

    ~~$x = d(e(x))$~~

    ~~Where x is a message, $d(\cdot)$ is a decoder, and $e(\cdot)$ is an encoder. Bob can~~
  - "Network sniffer"

- **RSA is based on which basic property of modulo arithmetic?**

  - The basic feature of modulo arithmetic that RSA exploits is the dramatic contrast in the tractibility of factoring and primality testing.
    - * Bob and Alice only need to make simple calculations.
    - * Eve needs to make computations that would be struggle for the world's largest computers.

$(m^e)^d \equiv m \mod N$

  - Pick any two prime p and q and let $N = pq$. For any e relatively prime to $(p-1)(q-1)$:

$$(x^e)^d \equiv x \mod N$$

$N = pq$

m → message

e → private key

d → public key

- **Describe the steps of the RSA protocol. The security of the protocol is based** on which assumption?

  Given N, e, and $y = x^e \mod N$, it is computationally intractable to determine x.

- **What are the basic operations that need to be performed accords to the RSA protocol and what is their running time?**

  - Pick two large primes $O(\log n)$
  - Multiply two large primes $O(n^2)$
  - Extended Euclid algorithm
  - Efficient modular exponentian algorithm
  - $y^d \mod N$

- **You may be provided an example message and asked to described the operations of the RSA protocol on it.**

- **What does Fermat's Little Theorem specify? Prove it.**

  - **Fermat's Little Theorem**: If p is a prime number, then for any integer a, the number $a^p - a$ is an integer multiple of p.
  - **Proof**

  - * Let us assume that a is positive and not divisible by p. The idea is that if we write down the sequence of numbers

$$a2a, 3a, \cdots, (p-1)a$$

and reduce each one modulo p, the result sequences turns out to be an arrangment of

$$1, 2, 3, \cdots, p-1$$

Therefore, if we multiple together the numbers in each sequences, the results must be indetifical modulo p:

$$a \times 2a \times 3a \times \cdots \times (p-1)$$

Which is equivilent to

$$1 \times 2 \times 3 \times \cdots \times (p-1)$$

Collecting the a terms yields

$$a^{p-1}(p-1)! \equiv (p-1)!(\mod p)$$

Finally, we may "cancel out" the numbers $1, 2, \cdots, p-1$ from both sides, obtains,

$$a^{p-1} \equiv 1(\mod p)$$

- **What is the importance of Fermat's little theorem in the RSA protocol?**

  – It's used to prove it.

  The proof of the correctness of RSA is based on Fermat's little theorem. This theorem states that if p is prime and p does not divide an integer a then

$$a^{p-1} \equiv 1 \pmod p$$

We want to show that $(m^e)^d \equiv m(\mod pq)$ for every integer m when p and q are distinct prime numbers and e and d are positive integers satisfying

$$ed \equiv 1 \pmod{(p-1)(q-1)}$$

We can write

$$ed - 1 = h(p-1)(q-1)$$

for some nonnegative integer h.

To check two numbers, like m^{ed} and m, are congruent mod pq it suffices (and in fact is equivalent) to check they are congruent mod p and mod q separately. (This is part of the Chinese remainder theorem, although it is not the significant part of that theorem.) To show $m^{ed} \equiv m(\text{mod} p)$, we consider two cases: $m0(modp)$

In the first case m^{ed} is a multiple of p, so $m^{ed} \quad 0 \quad m$ (mod p). In the second case

$$m^{ed} = m^{(ed-1)}m = m^{h(p-1)(q-1)}m = \left(m^{p-1}\right)^{h(q-1)} m \equiv 1^{h(q-1)}m \equiv m \pmod{p}$$

where we used Fermat's little theorem to replace m^{p-1} mod p with 1.

The verification that $m^{ed} \quad m$ (mod q) proceeds in a similar way, treating separately the cases $m \quad 0$ (mod q) and m 0 (mod q), using Fermat's little theorem for modulus q in the second case.

This completes the proof that, for any integer m,

$$\left(m^{e}\right)^{d} \equiv m \pmod{pq}$$

- **How can we efficiently perform modular exponentiantion? What is the running time of the approach?**
  - Recursion, repeated squaring.
  - Let n be the size of bits x, y, and N (whichever is largest).
  - Like multiplication, there are *at most* n recurisve calls.
  - During each call, it multiples n-bit numbers.
  - Doing computation modulo N saves us here.
  - Running time of $O(n^3)$

Related exercises

- DPV: 1.17, 1.27, 1.28, 1.35, 1.39, 1.42, 1.43, 1.44

Greatest Common Divisor algorithms: Euclid's test, Modulo Multiplicative Inverse

Reading material

- Chapters 1.2 from DPV
- Chapter 31.2 from CLRS2

Practice questions

- **What does Euclid's rule specify? Prove it.**
    - **Euclid's rule**: If x and y are positive integers where $x \leq y$, then $\gcd(x, y) = \gcd(x \bmod y, y)$
    - **Proof**
        * Suppose $a, b \in \mathbb{Z}$ and $a \vee B \neq 0$
        * From the *Division Theorem,a = qb + r\$ where $0 \leq r < |b|$.
        * From *GCD with Remainder*, the GCD of a and b is also the GCD of b and r.
        * Therefore, we may search instead for $\gcd(b, r)$.
        * Since $|r| < |b|$ and $b \in \mathbb{Z}$, we will reach $r = 0$ after finitely many steps.
        * At this point, $\gcd(r, 0) = r$ from *GCD with Zero*.
    - **Proof *from the book***:
        * It is enough to show the slightly simpler rule:

          $$\gcd(x, y) = \gcd(x - y, y)$$

          from which the one stated can be derived by repeatedly subtracting y from x.
        * Any integer that divides both x and y must also divide $x - y$, so

          $$\gcd(x, y) \leq \gcd(x - y, y)$$

        * Likewise, any integer that divides both $x - y$ and y must also divide both x and y, so

          $$\gcd(x, y) \geq gcd(x - y, y)$$

- **What is Euclid's algorithm for finding the greatest common divisor? What is its running time and why?** ~ function euclid(a, b) { if (b == 0) { return a; }

  ```
  else {
      return euclid(b, a % b);
  }
  ```

  } ~

    - It is, "If d divides both a and b, and $d = ax + by$ for some integers x and y, then necessarily $d = \gcd(a, b)$."
    - **Run time**
        * Let $T(a, b)$ be the number of steps taken in the Euclidean algorithm.

- Let $h = \log_{10} b$ be the number of digits in b.
- Assume that the modulo function is $O(1)$.
- The worst case is consecutive Fibonacci numbers.

$$a = F_{n+1}, b = F_n$$

- The algorith will calcuate the following until $n = 0$,

$$\gcd(F_{n+1}, F_n) = \gcd(F_n, F_{n-1})$$

So,

$$T(F_{n+1}, F_n) = \Theta(n)$$
$$T(a, F_n) = O(n)$$

- Since $F_n = \Omega(\Phi^n)$, this implies that

$$T(a, b) = O(\log_\Phi b)$$

- Note that $h \approx log_{10}b$ and $log_bx = \frac{\log x}{\log b}$, this implies $\log_b x = O(\log x)$.
- So the worst case for Euclid's algorith is

$$O(\log_\Phi b) = O(h) = \log(b)$$

- **Show that if d divides both a and b, and $d = a \times x + b \times y$ for some integers x and y, then $d = \gcd(a, b)$.**

    - By the first two condition, d is a common divisor of a and b so it cannot exceed the GCD, that is, $d \leq \gcd(a, b)$.
    - On the other hand, $\gcd(a, b)$ is a common divisor of a and b, it must also divide $ax + by = d$, which implies that $\gcd(a, b) \leq d$.
    - Therefore, $d = \gcd(a, b)$.

- **What is the extended Euclid's algorithm for finding the greatest common divisor d of two numbers a and b, as well as numbers x and y, so that $d = a \times x + b \times y$? Prove its correctness (you will need to prove Euclid's algorithm first and then the extension).**

    EEA: If d divides both a and b, and $d = ax + by$ for some integers x and y, then $d = \gcd(a, b)$

    - *Proof*
        - By the first two condition, d is a common divisor of a and b so it cannot exceed the GCD, that is, $d \leq \gcd(a, b)$.
        - On the other hand, $\gcd(a, b)$ is a common divisor of a and b, it must also divide $ax + by = d$, which implies that $\gcd(a, b) \leq d$.

        - Therefore, $d = \gcd(a, b)$.

- **When does the multiplicative inverse of a number x exists modulo N and why?**
- **How can you compute the multiplicative inverse modulo N for two relative prime numbers? What is the running time of the corresponding algorithm?**

Related exercises

- DPV: 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.33, 1.37

Primality Testing and Universal Hashing

Reading material

Chapter 1.3 from DPV

- Is there a test to tell us whether a number is prime? We place our home in a theorom from 1640.

Fermat's Little Theorem If p is prime, then $\forall 1 \leq a < p$,

$$a^{p-1} \equiv 1 (\text{mod} p)$$

Chapter 1.5 from DPV

- Chapters 31.8, 11.2, 11.3 from CLRS2

Practice questions

- **Describe a test for evaluating whether a number is prime. What is the probability of this test returning the correct answer and why? Can you increase the probability of success for this test?**
    - Fermat's little theorem states that if p is prime and $1 \leq a < p$, then,

    $$a^{p-1} \equiv 1 (\text{mod} p)$$

    - If we want to test if p is prime, then we can pick random a's in the interval and see if the equality holds.
        - If the equlaity *doesn't* hold for a value, then p is composite.

    - If the equality *does* hold for *many* values of a, the p is *probable prime.*
  - Using fast algorithms for modular exponentiation, the running time of this algorith is

    $$O(k \times \log^2 n \times \log\log n \times \log\log\log n)$$

    Where k is the number of time we test a random a, and n is the value we want to test for primality.
  - You increase the probability with a larger k.

- **What does Lagrange's Prime Number Theorem specify and why is it helpful for primality testing?**

  Lagrange's prime number theorem Let $\pi(x)$ be the number of prime $\leq x$. Then $\pi(x) \equiv x/(\ln x)$, or more precisely,

  $$\lim_{x \to \infty} \frac{\pi(x)}{(x/\ln x)} = 1$$

  - Such abundance makes it simple to generate a random n-bit prime:
    - Pcik a random n-bit number N.
    - Run a primality test on N.
    - If it passes the test, output N; else repeat the process.

- **What properties should a good hash function provide?**

  - Wikipedia
    - Determinism, same input same output
    - Uniformity, map the expected inputs as evenly as possible.
    - Variable range, can be expanded or contracted in size.
    - Continuity, two similar inputs should be mapped to nearly equal hashes
  - CLRS2
    - A good hash function satisifes the assumption of simple uniform hasing: each key is equally likely to hash to any of the m slots, indepedantly of where any other key has hashed to.

- **What is the property of universal hashing families of functions? Provide a universal hashing family of functions for an example problem and prove the corresponding property.**

  - Universal hashing is designed to stop the effectiveness of a attacker who knows the hash function and wants to put every input in the same bin.

- To avoid this, at the beginning of the lifecycle of a hash table, a function is picked *at random*.

Property Consider any pair of distinct IP addresses $x = (x_1, \cdots, x_4)$ and $y = (y_1, \cdots, y_4)$. If the coefficients $a = (a_1, a_2, a_3, a_4)$ are chosen uniformly at random from $\{0, 1, \cdots, n-1$ then

$$Pr\{h_a(x_1, \cdots, x_2) = h_a(y_1, \cdots, y_4)\} = \frac{1}{n}$$

- *Proof*
  - * Since $x = (x_1, \cdots x_4)$ and $y = (y_1, \cdots, y_4)$ are distinct, these quadruples must differ in some component, assume their not equal.
  - * We wish to compute the probability $Pr[h_a(x_1, \cdots, x_4) = h_a(y_1 \cdots y_4)]$
  - * That is, that

$$\sum_{i=1}^{4} a_i \times x_i \equiv \sum_{i=1}^{4} a_i \times y_i \bmod n$$

  - * This last equation can be rewritten

$$\sum_{i=1}^{3} a_i \times (x_i - y_i) \equiv a_4 \times (y_4 - x_4) \bmod n$$

  - * Suppose that we draw a random hash function h_a by picking $a = (a_1, a_2, a_3, a_4)$ at random.
  - * We start by drawing a_1, a_2, a_3, and then pause to think: What is the probability that the last drawn number a_4 is such that the equation 2 bullet points back holds (call it (1))?
  - * So far the left-hand side of equation (1) evaluates to some number, say c.
  - * And since n is prime and $x_4 \neq y_4$, $(y_4 - x_4)$ has a unique inverse modulo n.
  - * Thus for equation (1) to hold, the last number a_4 must be precisely $c \times (y_4 - x_4)^{-1} \bmod n$, out of its n possible values.
  - * The probability of this happening is $\frac{1}{n}$, and the proof is complete.

Related exercises

- DPV: 1.29, 1.34, 1.35

Divide and Conquer Algorithms and Recurrence Functions, Mergesort

Reading material

- Chapter 2.1-2.2-2.3 from DPV
- Chapters 2.3,4.1,4.2,4.3 from CLRS2

Practice questions

- **What are the basic principles of divide-and-conquer algorithms?**

  1. Breaking it into *subproblems* that are themselves smaller instancs of the same type of problem.
  2. Recursively solving these subproblems.
  3. Appropriately combining their answers.

- **Describe an approach for multiplication of two n-bit numbers that has a better running time than $O(n^2)$. Prove its running time.**

```
function multiply(x, y) {
    n = max(sizeof(x), sizeof(y));

    if (n == 1) {
        return xy;
    }

    x_l = leftmostBits(ceil(n / 2));
    x_r = rightmostBits(floor(n / 2));
    y_l = leftmost(ceil(n / 2));
    y_r = rightmostBits(floor(n / 2));

    p_1 = multiply(x_l, y_l);
    p_2 = multiplu(x_r, y_r);
    p_3 = multiply(x_l + x_r, y_l + y_r);

    return p_1 x 2^n + (p_3 - p_1 - p_2) x 2^(n / 2) + p_2;
}
```

  - Run time
    - This running time can be derived by looking at the algorithm's pattern of recursive calls, which form a tree-structure.
    - At aech successive level of recursion the subproblems get halved in size. At the log_2nth level, the subproblems get down to size 1, and so the recursion ends.

        - Therefore, the high of the tree is $\log_2 n$.
        - The branching factor is 4, each problem recursive produces three smaller ones - with the result that at depth k in the tree there are 3^k subproblems, each of size $n/2^k$.

- **What does the Master theorem specify? Prove it.** > **Master theorem**: If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some > constants $a > 0, b > 1, d \geq 0$, then: >

$$\begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < lob_b a \end{cases}$$

    - Proof
        1. Assume that n is a power of b, for convinience rounding.
        2. Notice that the size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $log_b n$ levels. This is the hight of the tree.
        3. Its branching factor is a, so the kth level of the tree is made up of a^k subproblems, each of size n/b^k. The total work is done:

$$a^k \times O(\frac{n}{b^k})^d = O(n^d) \times (\frac{a}{b^d})^k$$

        4. As k goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with the ration a/b^d. Finding the sum of such a series in big-O notation is easy and comes down to three cases:
            1. The ratio is less than 1.
            2. The ratio is greater than 1.
            3. The ratio is exactly 1.

- **You may be provided a divide-and-conquer algorithm and asked to argue about its running time by using the Master theorem.**

- **How does mergesort work, what is its running time and why? How does the iterative version of mergesort work?**

  Mergesort splits an unsorted array into two and sorts and concatenates the subarrays.

```
function interative-mergesort(a[1 .... n]) {
    Q = [] (empty queue);
    for i = 1 to n:
         inject(Q, a[i])l

    while(count(Q) > 1):
         inject(Q, merge(eject(Q), eject(Q));
    return eject(Q);
}
```

Related exercises

- DPV: 2.3, 2.4, 2.5, 2.12, 2.13, 2.14, 2.19, 2.25, 2.26, 2.31
- CLRS2: 2.3-1, 2.3-3, 4.3-1, 4.3-2, 4.3-3, 4.3-4
- CLRS2: 4-1, 4-2, 4-3, 4-4

Quicksort, Lower bounds for comparison-based sorting

Reading material

- Chapter 7.1, 7.2, 7.3, 7.4, 8.1 from CLRS2
- Chapter 2.3 from DPV

Practice questions

- **What are comparison sorting algorithms? What is the lower limit for the running time of comparison sorting algorithms?**
- **How does quick-sort work? When does the worst-case running time arise? When does the best-case running time arise?**

  1. Pick an element, called a pivot, from the list.
  2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
  3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

  The best case for the algorithm now occurs when all elements are equal. The worst case for the algorithm occurs when the elements are already sorted.

- **Provide a rigorous proof for the worst-case performance of quick-sort.**
- **Provide a rigorous proof for the expected running time of quick-sort.**

Related exercises

- CLRS2: 7.1-4, 7.2-2, 7.2-3, 7.2-4, 7.2-5, 7.3-1, 7.3-2, 7.4-1, 7.4-2, 7.4-3, 7.4-4, 7.4-5, 8.1-1, 8.1-3
- CLRS2: 7-3, 7-4, 7-5
- DPV: 2.17, 2.18, 2.24

Computing Medians and Other Order Statistics, Linear-time Sorting Solutions

Reading material

- Chapters 2.4 from DPV
- Chapters 8.2, 8.3, 8.4, 9.1, 9.2, 9.3 from CLRS2

Practice questions

- **How can we efficiently compute the median of a set of numbers? What is the running time of this solution?**

- **What is the main idea behind counting sort? Provide a description of the algorithm and argue about its running time.**

  > the algorithm loops over the items, computing a histogram of the number of times each key occurs within the input collection. It then performs a prefix sum computation (a second loop, over the range of possible keys) to determine, for each key, the starting position in the output array of the items having that key. Finally, it loops over the items again, moving each item into its sorted position in the output array.

```
# variables:
#   input -- the array of items to be sorted; key(x) returns the key for item x
#    n -- the length of the input
#    k -- a number such that all keys are in the range 0..k-1
#   count -- an array of numbers, with indexes 0..k-1, initially all zero
#    output -- an array of items, with indexes 0..n-1
#    x -- an individual input item, used within the algorithm
#    total, oldCount, i -- numbers used within the algorithm

# calculate the histogram of key frequencies:
for x in input:
    count[key(x)] += 1
```

```
# calculate the starting index for each key:
total = 0
for i in range(k):   # i = 0, 1, ... k-1
    oldCount = count[i]
    count[i] = total
    total += oldCount

# copy to output array, preserving order of inputs with equal keys:
for x in input:
    output[count[key(x)]] = x
    count[key(x)] += 1

return output
```

Because the algorithm uses only simple for loops, without recursion or subroutine calls, it is straightforward to analyze. The initialization of the Count array, and the second for loop which performs a prefix sum on the count array, each iterate at most k + 1 times and therefore take O(k) time. The other two for loops, and the initialization of the output array, each take O(n) time. Therefore the time for the whole algorithm is the sum of the times for these steps, O(n + k).

Because it uses arrays of length k + 1 and n, the total space usage of the algorithm is also O(n + k).[1] For problem instances in which the maximum key value is significantly smaller than the number of items, counting sort can be highly space-efficient, as the only storage it uses other than its input and output arrays is the Count array which uses space O(k).

- **What is the main idea behind radix sort? Provide a description of the algorithm and argue about its running time.**

  Radix sort works by first sorting by the least significant bit, then the next most significant bit, all the way to the most significant bit.

- **Prove the correctness of radix sort.**

  For a value $r \leq b$, we view each key as having $d = \lceil b/r \rceil$ digits of r bits each. Each digit is an integer in the range of 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$. Each pass of counting sort takes time $\Omega(n + k) = \Omega(n + 2^r)$ and there are d passes. This makes a total running time of $\Omega(d(n + 2^r))$.

- **What is the main idea behind bucket sort? Provide a description of the algorithm. Prove its running time.**

  - Idea

    1. Set up an array of initially empty “buckets”.

merge
divide|conquer
splits

Radix
least significant
bit.

bucket
split, handle
buckets, combine

Counting
histogram,
index in.

2. Scatter: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. Gather: Visit the buckets in order and put all elements back into the original array.

   The idea behind bucket sort is that if we know the range of our elements to be sorted, we can set up buckets for each possible element, and just toss elements into their corresponding buckets. We then empty the buckets in order, and the result is a sorted list.

- Psuedocode

```
function bucketSort(array, n) is
    buckets ← new array of n empty lists
    for i = 0 to (length(array)-1) do
        insert array[i] into buckets[msbits(array[i], k)]

    for i = 0 to n - 1 do
        nextSort(buckets[i]);

  return the concatenation of buckets[0], ...., buckets[n-1]
```

Related exercises

- DPV: 2.15, 2.16, 2.17, 2.20, 2.21, 2.22, 2.23
- CLRS2: 8.2-2, 8.2-4, 8.3-2, 8.3-3, 8.3-4, 8.4-2, 8.4-3, 9.1-1, 9.3-5, 9-3.7, 9.3-8, 9.3-9
- CLRS2: 8-2, 8-3, 8-6, 9-1, 9-2, 9-3

Greedy Algorithms: Huffman encoding

Reading material

Chapter 5.2 DPV

- In general, how do we find the optimal coding tree, given the frequencies $f_1 \cdots f_n$ of n symbols? To make the problem precise, we want a tree whose leaves each correspond to a symbol and which minimizes the overall length of the encoding. **Cost of tree**, where d is the depth of the ith symbol in the tree:

$$\sum_{i=1}^{n} f_i \times d$$

- We can define the frequency of any internal node to be the sum of the frequencies of its descendant leaves; this is, after all, the number of times the internal node is visit during encoding or decoding.
  - The total cost can be expressed thusly:

    > The cost of a tree is the sum of the frequences of all leaves and internal nodes, except the root.

```
function Huffman(f)
input: Any array f[1 ... n] of frequencies
output: An encoding tree with n leaves

let H be a priority queue of integers, ordered b f
for i = 1 to n: insert(H, i)
for k = n + 1 to 2n - 1:
    i = deletemin(h), j = deletemin(h)
    create a node numbered k with children i, j
    f[k] = f[i] + f[j]
    insert(H, k)
```

Chapter 16.2 from CLRS2

Greedy algorithm A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

- Greedy algorithms do not always yield optimal solution, but for many problems they do.
- The following steps apply to a greedy algorithm strategy:
  1. Determine the optimal substructure of the problem.
  2. Develop a recursive solution.
  3. Prove that at any stage of the recursion, one of the optimal choices is the greedy choise. Thus, it always safe to make the greedy choice.
  4. Show that all but one of the subproblems induced by having made the greedy choice are empty.
  5. Develop a recursive algorithm that implements the greedy strategy.
  6. Convert the recursive algorithm to an iterative algorithm.

Chapter 16.3 from CLRS2

Wikipedia

- A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.
  - For instance, to solve the travelling salesman problem with a greedy strategy, adopt the following strategy: "At each stage visit an unvisited city nearest to the current city."
- There are five components:
  1. A candidate set, from which a solution is created.
  2. A selection function, which chooses the best candate to be added to the solution.
  3. A feasibility function, that is used to determine if a candidate can be used to contribute to a solution.
  4. An objective function, which assigns a value to a solution, or a partial solution, and
  5. A solution function, which will indicate when we have discovered a complete solution.

Practice questions

Greedy algo pick local optimum hoping its global too.

- **What is the main principle behind greedy algorithms?**
  - The main principle behind greedy algorithms is to always, in any given decision point, to select the locally optimal choice and hope that it leads to a globally optimal choice (which has no guarentee).
- **Why does the greedy algorithm work for the coin changing problem given the US coin system?**
  - The US coin system placed in a set of integers has the property of being a **matroid**.
- **What is the idea in Huffman encoding in order to achieve data compression? What is the prefix-free property?**
  - Huffman oding is an entropy encoding algorithm used for lossless data compression.
  - The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurance for each possible value of the source symbol.

- **Provide the Huffman encoding algorithm and argue its running time.**

    1. Create a leaf node for each symbol and add it to the priority queue.
    2. While there is more than one node in the queue:
        1. Remove the two nodes of highest priority (lowest probability) from the queue.
        2. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes probabilities.
        3. Add the new node to the queue.
    3. The remaining node is the root node and the tree is complete.

- **Prove the greedy choice property for the Huffman encoding algorithm (first lemma).**

    > ***Lemma 16.2*** Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two character in C having the lowest frequencies. Then there exists an optimal prefix code C in which the codewords for x and y have the same length and differ only in the last bit.

- **Prove the optimal substructure property for the Huffman encoding algorithm (second lemma).**

Related exercises

- DPV: 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, 5.19, 5.29, 5.30, 5.31
- CLRS2: 16.1, 16.2-4, 16.2-5, 16.2-7, 16.3-1, 16-3.2, 16.3-3, 16.3-4, 16.3-5, 16.3-6