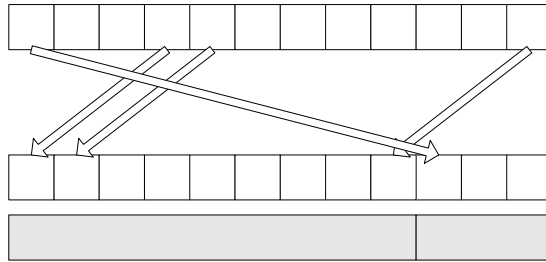


CS 432 Algorithms, Fall 2004

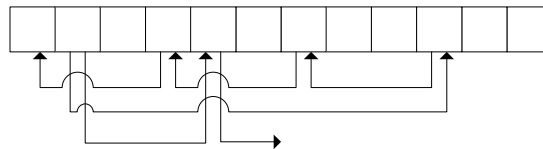
Problems Set 1: Solutions to Optional Problems

Problem B1

Notice that the character that ends up in position j comes from position $(j + i)$ when $j \leq (n - i)$ and $(j + i - n)$ when $j > (n - i)$:



If we use a buffer to store X_1 temporarily we have a 'hole' into which we can move X_{i+1} . Once we've done that X_{i+1} becomes a hole into which we can move X_{i+2} , and so on:



So in pseudocode:

```
function rotate_X_left_by_i(vector X1..n, integer i)
  to := 1, from := i + 1
  scratch := X1
  while from ≠ 1
    Xto := Xfrom
    to := from
    if (to + i) ≤ n then from := to + i else from := to + i - n
  end rotate_X_left_by_i
```

There are other ways to implement the rotation. For example, it is possible to write a function to reverse a vector in space complexity $\Theta(1)$, so this solution also runs in space complexity $\Theta(1)$:

```
function rotate(vector X1..n, integer i)
  reverse(X1..i)
  reverse(Xi+1..n)
  reverse(X1..n)
end rotate
```

if X was "Hello, world" and i=3
 # X is now "leHlo, world"
 # X is now "leHdlrow ,ol"
 # X is now "lo, worldHel". Yay!

1 2 3

Problem B2

Start by tagging each word in the dictionary with a vector containing the number of A's, B's, C's, and so on, so that "cab" would be tagged with $\langle 1, 1, 1, 0, 0, \dots \text{many } 0\text{'s} \dots \rangle$, "babe" would be tagged with $\langle 1, 2, 0, 0, 1, 0, \dots \text{many } 0\text{'s} \dots \rangle$, and so forth. A word is then an anagram of another word if

H @ 1

they both have the same tag. This can be done in constant time. (Well, constant in terms of the number of words in the dictionary, anyway. This does depend upon the length of each word, but that's upper bounded by a reasonably small constant.)

Sort the dictionary according to the tag values. As the prompt states, this can be done in $\Theta(n \log n)$ time.

Traverse the dictionary looking for sets of words with the same tag value. Since the dictionary has been sorted by the tag values, anagrams are grouped together and we can do this in constant time.

Since the most expensive part of this algorithm is $\Theta(n \log n)$, the algorithm is $\Theta(n \log n)$.

There are other solutions to this. One is to use a copy of the word with the characters sorted by value as the key. In this scheme, the keys for "rat", "tar", and "art" would all be "art".

Problem B3

Part A: Divide and conquer solution.

```
function max_subvect(vector X0..n) {
  # an empty subvector has value 0
  if n < 0 then return 0

  # a one-element vector's max is either its element or the empty subvector
  if n = 0 then return max(0, X0)

  # divide and conquer
  pivot := midpoint between 0 and n
  left := max_subvect(X0..pivot-1)
  right := max_subvect(Xpivot+1..n)

  # but what if the maximum subvector spans the midpoint?
  centre := max_subvect_spanning_centre(X0..n, pivot)

  return max(left, right, centre)
end max_subvect

function max_subvect_spanning_centre(vector X0..n, position centre)
  left := 0, right := 0, sum := 0

  # find the largest subvector that ends just to the left of centre
  for i from centre - 1 to 1 by -1
    sum := sum + Xi
    left := max(left, sum)

  # now do the same for the right
  for i from centre + 1 to n
    sum := sum + Xi
    right := max(right, sum)

  return left + right + Xcentre
end max_subvect_spanning_centre
```

Taken together these functions give the recurrence relationship $T(n) = 2T(\frac{n}{2}) + n$, which is $O(n \log n)$.

Part B: On-line solution.

```
function max_subvect(vector X0..n)
  biggest := 0, max_ending_here := 0
  for i from 0 to n
    max_ending_here := max(max_ending_here + Xi, 0)
    biggest := max(biggest, max_ending_here)
  return biggest
end max_subvect
```

This is on-line because it accesses each X_i exactly once in order from 0 to n . This solution's loop executes $n+1$ times performing a constant amount of work each time, so it is $\Theta(n)$.

Problem O1:

If there is exactly one fake coin, you can find it by:

1. Dividing the pile of n coins into two piles of $\lfloor \frac{n}{2} \rfloor$ coins. If n is odd there will be one extra coin.

If there is only an extra coin—because n was 1—then it is the fake.

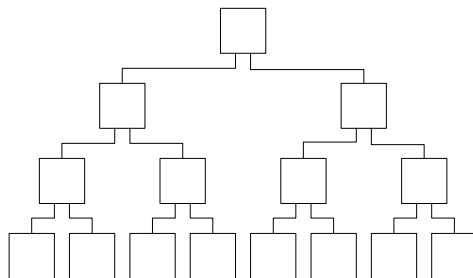
2. Put the piles on the balance. If they have the same weight, the extra coin must be the fake. If not, repeat these steps on the lighter pile until the fake is identified.

This algorithm identifies the fake in at most $\lfloor \log_2 n \rfloor$ weighings.

If there is either 0 or 1 fake coins, the algorithm needs a slight modification: in step 2 we can't assume that if the piles have the same weight the extra coin is fake. If the two piles have the same weight, substitute the extra coin for a coin from one of the piles. If the piles still have the same weight there is no fake coin; if not then the extra was indeed fake.

Problem O2 part A:

Here's a sample tournament tree for $2^3 = 8$ teams:



Note that in every round we play half as many games as there are teams, and half as many teams continue on to the next round. The total number of games played is:

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1.$$

Problem O3:

We can show by induction that for all $n \geq 1$ $\frac{a^n}{b^n} < 1$:

First, note that for $n = 1$: $\frac{a^n}{b^n} = \frac{a}{b} < 1$ since $1 < a < b$.

Next, note that for $n > 1$: $\frac{a^n}{b^n} = \left(\frac{a}{b}\right) \left(\frac{a^{n-1}}{b^{n-1}}\right) < 1$ since $0 < \frac{a}{b} < 1$ and $0 < \frac{a^{n-1}}{b^{n-1}} < 1$.

Since we know that for all $n \geq 1$, $\frac{a^n}{b^n} < 1$, we know that there are constants $N(1)$ and C (also 1) such that for all $n \geq N$, $a^n < Cb^n$, so $a^n = o(b^n)$.

Problem O4:

$$\lim_{n \rightarrow \infty} \frac{n^k}{\lg n} = \lim_{n \rightarrow \infty} \frac{n^k \ln 2}{\ln n} = \frac{\infty}{\infty}$$

Applying L'Hopital's rule once gives us:

$$\lim_{n \rightarrow \infty} \frac{n^k}{\lg n} = \lim_{n \rightarrow \infty} \frac{kn^{k-1} \ln 2}{\ln n} = \frac{\infty}{\infty}$$

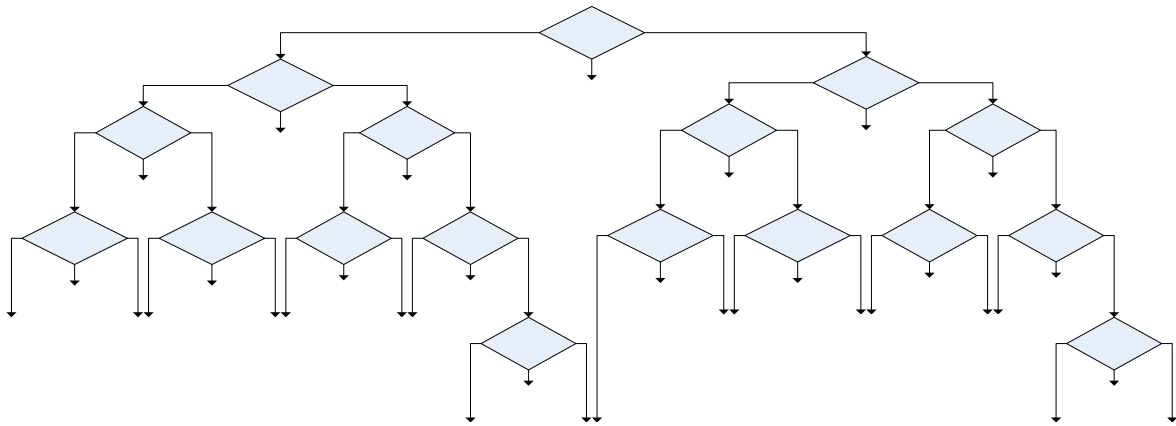
If k is an integer and we apply L'Hopital's rule repeatedly we eventually reach:

$$\lim_{n \rightarrow \infty} \frac{n^k}{\lg n} = \frac{k! \ln 2}{\ln n} = 0 \text{ (since } k! \text{ and } \ln 2 \text{ are constants)}$$

Thus $\lg n = o(n^k)$ for any $k > 0$.

Problem O6:

Here it is, drawn with the key shown as κ and the array to be searched as $X_{0..16}$. My apologies for the small type...



Problem R1 parts B, C, and D:

Part B: If we let $a = 1$, $b = 2$, and $k = 1$ then $T(n) = T\left(\frac{n}{2}\right) + n = aT\left(\frac{n}{b}\right) + f(n^k)$. $a < b^k$ so by the main recursion theorem $T(n) = \Theta(n)$.

Part C: If we let $a = 2$, $b = 2$, and $k = 1$ then $T(n) = 2T\left(\frac{n}{2}\right) + n = aT\left(\frac{n}{b}\right) + f(n^k)$. $a = b^k$ so by the main recursion theorem $T(n) = \Theta(n \log n)$.

Part D:

Problems 4S, 7S, and 10S have solutions in the text.

Problem 11:

Factoring $\frac{(n^2 + \lg n)(n+1)}{n+n^2}$ this yields $\frac{n^3 + n^2 + n \lg n + \lg n}{n+n^2}$. The n^3 term dominates the numerator and the n^2 dominates the denominator, so as n increases this expression tends toward n , so it is $\Theta(n)$.

Problem 16S has a solution in the text.

Problems 23S and 38S have solutions in the text.