4 Randomization

As deterministic algorithms are driven to their limits when one tries to solve hard problems with them, a useful technique to speed up the computation is **randomization**. In randomized algorithms, the algorithm has access to a *random source*, which can be imagined as tossing coins during the computation. Depending on the outcome of the toss, the algorithm may split up its computation path.

There are two main types of randomized algorithms: Las Vegas algorithms and Monte-Carlo algorithms. In Las Vegas algorithms, the algorithm may use the randomness to speed up the computation, but the algorithm must always return the correct answer to the input. Monte-Carlo algorithms do not have the former restriction, that is, they are allowed to give *wrong* return values. However, returning a wrong return value must have a *small probability*, otherwise that Monte-Carlo algorithm would not be of any use.

Many approximation algorithms use randomization.

4.1 Ordered Statistics

Before covering randomized techniques, we'll start with a deterministic problem that leads to a problem that utilizes randomization. Suppose you have an unsorted array of values and you want to find

- the maximum value,
- the minimum value, and
- the median value.

In the immortal words of one of our former computer science professors, "How can you do?"

4.1.1 find-max

First, it's relatively straightforward to find the largest element:

```
// find-max -- returns the maximum element
function find-max(array vals[1..n]): element
  let result := vals[1]
  for i from 2 to n:
    result := max(result, vals[i])
  repeat

  return result
end
```

An initial assignment of $-\infty$ to *result* would work as well, but this is a useless call to the max function since the first element compared gets set to *result*. By initializing result as such the function only requires *n-1* comparisons. (Moreover, in languages capable of metaprogramming, the data type may not be strictly numerical and there might be no good way of assigning $-\infty$; using vals[1] is type-safe.)

A similar routine to find the minimum element can be done by calling the min function instead of the max function.

4.1.2 find-min-max

But now suppose you want to find the min and the max at the same time; here's one solution:

```
// find-min-max -- returns the minimum and maximum element of the given array
function find-min-max(array vals): pair
  return pair {find-min(vals), find-max(vals)}
end
```

Because **find-max** and **find-min** both make *n-1* calls to the max or min functions (when *vals* has n elements), the total number of comparisons made in **find-min-max** is $2n - 2$.

However, some redundant comparisons are being made. These redundancies can be removed by "weaving" together the min and max functions:

```
// find-min-max -- returns the minimum and maximum element of the given array
function find-min-max(array vals[1..n]): pair
  let min := ∞
  let max := −∞

  if n is odd:
    min := max := vals[1]
    vals := vals[2,..,n]          // we can now assume n is even
    n := n - 1
  fi

  for i:=1 to n by 2:             // consider pairs of values in vals
    if vals[i] < vals[i + n by 2]:
      let a := vals[i]
      let b := vals[i + n by 2]
    else:
      let a := vals[i + n by 2]
      let b := vals[i]            // invariant:  a <= b
    fi

    if a < min: min := a fi
    if b > max: max := b fi
  repeat

  return pair {min, max}
end
```

Here, we only loop $n/2$ times instead of n times, but for each iteration we make three comparisons. Thus, the number of comparisons made is $(3/2)n = 1.5n$, resulting in a 3/4 speed up over the original algorithm.

Only three comparisons need to be made instead of four because, by construction, it's always the case that $a \leq b$. (In the first part of the "if", we actually know more specifically that $a < b$, but under the else part, we can only conclude that $a \leq b$.) This property is utilized by noting that a doesn't need to be compared with the current maximum, because b is already greater than or equal to a, and similarly, b doesn't need to be compared with the current minimum, because a is already less than or equal to b.

In software engineering, there is a struggle between using libraries versus writing customized algorithms. In this case, the min and max functions weren't used in order to get a faster **find-min-max** routine. Such an operation would probably not be the bottleneck in a real-life program: however, if testing reveals the routine should be faster, such an approach should be taken. Typically, the solution that reuses libraries is better overall than writing customized solutions. Techniques such as open implementation and aspect-oriented programming may help manage this contention to get the best of both worlds, but regardless it's a useful distinction to recognize.

4.1.3 find-median

Finally, we need to consider how to find the median value. One approach is to sort the array then extract the median from the position *vals*[*n*/2]:

```
// find-median -- returns the median element of vals
function find-median(array vals[1..n]): element
  assert (n > 0)

  sort(vals)
  return vals[n / 2]
end
```

If our values are not numbers close enough in value (or otherwise cannot be sorted by a radix sort) the sort above is going to require $O(n \log n)$ steps.

However, it is possible to extract the nth-ordered statistic in $O(n)$ time. The key is eliminating the sort: we don't actually require the entire array to be sorted in order to find the median, so there is some waste in sorting the entire array first. One technique we'll use to accomplish this is randomness.

Before presenting a non-sorting **find-median** function, we introduce a divide and conquer-style operation known as **partitioning**. What we want is a routine that finds a random element in the array and then partitions the array into three parts:

1. elements that are less than or equal to the random element;
2. elements that are equal to the random element; and
3. elements that are greater than or equal to the random element.

These three sections are denoted by two integers: j and i. The partitioning is performed "in place" in the array:

```
// partition -- break the array three partitions based on a randomly picked element
function partition(array vals): pair{j, i}
```

Note that when the random element picked is actually represented three or more times in the array it's possible for entries in all three partitions to have the same value as the random element. While this operation may not sound very useful, it has a powerful property that can be exploited: When the partition operation completes, the randomly picked element will be in the same position in the array as it would be if the array were fully sorted!

This property might not sound so powerful, but recall the optimization for the **find-min-max** function: we noticed that by picking elements from the array in pairs and comparing them to each other first we could reduce the total number of comparisons needed (because the current min and max values need to be compared with only one value each, and not two). A similar concept is used here.

While the code for **partition** is not magical, it has some tricky boundary cases:

```
// partition -- break the array into three ordered partitions from a random element
function partition(array vals): pair{j, i}
  let m := 0
  let n := vals.length - 1
  let irand := random(m, n)   // returns any value from m to n
  let x := vals[irand]
  // values in vals[n..] are greater than x
  // values in vals[0..m] are less than x
  while (m <= n)
     if vals[m] <= x
        m++
     else
        swap(m,n)                // exchange vals[n] and vals[m]
        n--
     endif
  endwhile
  // partition: [0..m-1]   []   [n+1..]   note that m=n+1
  // if you need non empty sub-arrays:
  swap(irand,n)
  // partition: [0..n-1]   [n..n]   [n+1..]
end
```

We can use **partition** as a subroutine for a general **find** operation:

```
// find -- moves elements in vals such that location k holds the value it would when sorted
function find(array vals, integer k)
  assert (0 <= k < vals.length)        // k it must be a valid index
  if vals.length <= 1:
    return
  fi

  let pair (j, i) := partition(vals)
  if k <= i:
    find(a[0,..,i], k)
  else-if j <= k:
    find(a[j,..,n], k - j)
  fi
  TODO: debug this!
end
```

Which leads us to the punch-line:

```
 // find-median -- returns the median element of vals
function find-median(array vals): element
  assert (vals.length > 0)

  let median_index := vals.length / 2;
  find(vals, median_index)
  return vals[median_index]
end
```

One consideration that might cross your mind is "is the random call really necessary?" For example, instead of picking a random pivot, we could always pick the middle element instead. Given that our algorithm works with all possible arrays, we could conclude that the running time on average for *all of the possible inputs* is the same as our analysis that used the random function. The reasoning here is that under the set of all possible arrays, the middle element is going to be just as "random" as picking anything else. But there's a pitfall in this reasoning: Typically, the input to an algorithm in a program isn't random at all. For example, the input has a higher probability of being sorted than just by chance alone. Likewise, because it is real data from real programs, the data might have other patterns in it that could lead to suboptimal results.

To put this another way: for the randomized median finding algorithm, there is a very small probability it will run suboptimally, independent of what the input is; while for a deterministic algorithm that just picks the middle element, there is a greater chance it will run poorly on some of the most frequent input types it will receive. This leads us to the following guideline:

> **Randomization Guideline:**
> If your algorithm depends upon randomness, be sure you introduce the randomness yourself instead of depending upon the data to be random.

Note that there are "derandomization" techniques that can take an average-case fast algorithm and turn it into a fully deterministic algorithm. Sometimes the overhead of derandomization is so much that it requires very large datasets to get any gains. Nevertheless, derandomization in itself has theoretical value.

The randomized **find** algorithm was invented by C. A. R. "Tony" Hoare. While Hoare is an important figure in computer science, he may be best known in general circles for his quicksort algorithm, which we discuss in the next section.

4.2 Quicksort

The median-finding partitioning algorithm in the previous section is actually very close to the implementation of a full blown sorting algorithm. Until this section is written, building a Quicksort Algorithm is left as an exercise for the reader.

[TODO: Quicksort Algorithm]

4.3 Shuffling an Array

```
    This keeps data in during shuffle
    temporaryArray = { }
    This records if an item has been shuffled
    usedItemArray = { }
    Number of item in array
    itemNum = 0
    while ( itemNum != lengthOf( inputArray) ){
        usedItemArray[ itemNum ] = false None of the items have been shuffled
        itemNum = itemNum + 1
    }
    itemNum = 0 we'll use this again
    itemPosition = randdomNumber( 0 --- (lengthOf(inputArray) - 1 ))
    while( itemNum != lengthOf( inputArray ) ){
        while( usedItemArray[ itemPosition ] != false ){
            itemPosition = randdomNumber( 0 --- (lengthOf(inputArray)
- 1 ))
        }
        temporaryArray[ itemPosition ] = inputArray[ itemNum ]
    }
    inputArray = temporaryArray
```

4.4 Equal Multivariate Polynomials

[TODO: as of now, there is no known deterministic polynomial time solution, but there is a randomized polytime solution. The canonical example used to be IsPrime, but a deterministic, polytime solution has been found.]

4.5 Skip Lists

[TODO: Talk about skips lists. The point is to show how randomization can sometimes make a structure easier to understand, compared to the complexity of balanced trees.] In order to compare the complexity of implementation, it is a good idea to read about skip lists and red-black trees ; the internet has the skip list's author paper spruiking skip lists in a very informed and lucid way, whilst Sedgewick's graduated explanation of red-black trees by describing them as models of 2-3 trees is also on the internet on the website promoting his latest algorithms book.

4.5.1 Recap on red-black trees

What Sedgewick does is to diagrammatically show that the mechanisms maintaining 2-3 tree nodes as being either 2 nodes (a node with 2 children and 1 value) or 3 nodes (2 values separating 3 children) by making sure 4 nodes are always split into three 2-nodes , and the middle 2-node is passed up into the parent node, which may also split. 2-3 tree nodes are really very small B-trees in behavior. The inductive proof might be that if one tries to load everything into the left side of the tree by say a descending sequence of keys as input, the tree still looks balanced after a height of 4 is reached, then it will looked balanced at any height.

He then goes to assert that a 2 node can be modeled as binary node which has a black link from its parent, and a black link to each of its two children, and a link's color is carried by a color attribute on the child node of the link; and a 3-node is modeled as a binary node which has a red link between a child to a parent node whose other child is marked black, and the parent itself is marked black ; hence the 2 nodes of a 3-node is the parent red-linked to the red child. A 4-node which must be split, then becomes any 3 node combination which has 2 red links, or 2 nodes marked red, and one node , a top most parent, marked black. The 4 node can occur as a straight line of red links, between grandparent, red parent, and red grand child, a zig-zag, or a bow , black parent, with 2 red children, but everything gets converted to the last in order to simplify splitting of the 4-node.

To make calculations easier, if the red child is on the right of the parent, then a rotation should be done to make the relationship a left child relationship. This left rotation[1] is done by saving the red child's left child, making the parent the child's left child, the child's old left child the parents right child (formerly the red child), and the old parent's own parent the parent of the red child (by returning a reference to the old red child instead of the old parent), and then making the red child black, and the old parent red. Now the old parent has become a left red child of the former red child, but the relationship is still the two same keys making up the 3-node. When an insertion is done, the node to be inserted is always red, to model the 2-3 tree behavior of always adding to an existing node initially (a 2-node with all black linked children marked black, where terminal null links are black, will become a 3-node with a new red child).

Again, if the red child is on the right, a left rotation is done. It then suffices to see if the parent of the newly inserted red node is red (actually, Sedgewick does this as a recursive post insertion check, as each recursive function returns and the tree is walked up again, he checks for a node whose left child is red, and whose left child has a red left child) ; and if so, a 4-node exists, because there are two red links together. Since all red children have been rotated to be left children, it suffices to right rotate at the parent's parent, in order to make the middle node of the 4- node the ancestor linked node, and splitting of the 4 node and passing up the middle node is done by then making left and right nodes black, and the center node red, (which is equivalent to making the end keys of a 4-node into two 2-nodes, and the middle node passed up and merged with node above it). Then , the above process beginning with "if the red child is on the right of the parent ..." should then be carried out recursively with the red center node, until all red nodes are left child's , and the parent is not marked red.

The main operation of rotation is probably only marginally more difficult than understanding insertion into a singly linked list, which is what **skip lists** are built on , except that the nodes to be inserted have a variable height.

4.5.2 Skip list structure

A skip list node is an array of singly-linked list nodes, of randomized array length (height) , where a node array element of a given index has a pointer to another node array element only at the same index (=level) , of another skip list node. The start skip list node is

1 `http://en.wikibooks.org/wiki/..%2FLeft%20rotation`

referenced as the header of the skip list, and must be as high as the highest node in the skip list , because an element at a given index(height) only point to another element at the same index in another node, and hence is only reachable if the header node has the given level (height).

In order to get a height of a certain level n, a loop is iterated n times where a random function has successful generated a value below a certain threshold on n iterations. So for an even chance threshold of 0.5, and a random function generating 0 to 1.0, to achieve a height of 4, it would be 0.5 ˆ 4 total probability or (1/2)ˆ4 = 1/16. Therefore, high nodes are much less common than short nodes , which have a probability of 0.5 of the threshold succeeding the first time.

Insertion of a newly generated node of a randomized height, begins with a search at the highest level of the skip list's node, or the highest level of the inserting node, whichever is smaller. Once the position is found, if the node has an overall height greater than the skip list header, the skip list header is increased to the height of the excess levels, and all the new level elements of the header node are made to point to the inserting node (with the usual rule of pointing only to elements of the same level).

Beginning with header node of the skip list ,a search is made as in an ordered linked list for where to insert the node. The idea is to find the last node which has a key smaller than insertion key by finding a next node's key greater than the inserting key; and this last smaller node will be the previous node in a linked list insertion. However, the previous node is different at different levels, and so an array must be used to hold the previous node at each level. When the smallest previous node is found, search is recommenced at the next level lower, and this continues until the lowest level . Once the previous node is known for all levels of the inserting node, the inserting node's next pointer at each level must be made to point to the next pointer of the node in the saved array at the same level . Then all the nodes in the array must have their next pointer point to the newly inserted node. Although it is claimed to be easier to implement, there is two simultaneous ideas going on, and the locality of change is greater than say just recursively rotating tree nodes, so it is probably easier to implement, if the original paper by Pugh is printed out and in front of you, and you copy the skeleton of the spelled out algorithm as pseudocode from the paper down into comments, and then implement the comments. It is still basically singly linked list insertion, with a handle to the node just before , whose next pointer must be copied as the inserted node's next pointer, before the next pointer is updated as the inserted node; but there are other tricky things to remember, such as having two nested loops, a temporary array of previous node references to remember which node is the previous node at which level ; not inserting a node if the key already exists and is found, making sure the list header doesn't need to be updated because the height of the inserting node is the greatest encountered so far, and making multiple linked list insertion by iterating through the temporary array of previous pointers.

4.5.3 Role of Randomness

The idea of making higher nodes geometrically randomly less common, means there are less keys to compare with the higher the level of comparison, and since these are randomly selected, this should get rid of problems of degenerate input that makes it necessary to do

tree balancing in tree algorithms. Since the higher level list have more widely separated elements, but the search algorithm moves down a level after each search terminates at a level, the higher levels help "skip" over the need to search earlier elements on lower lists. Because there are multiple levels of skipping, it becomes less likely that a meagre skip at a higher level won't be compensated by better skips at lower levels, and Pugh claims O(logN) performance overall.

Conceptually , is it easier to understand than balancing trees and hence easier to implement ? The development of ideas from binary trees, balanced binary trees, 2-3 trees, red-black trees, and B-trees make a stronger conceptual network but is progressive in development, so arguably, once red-black trees are understood, they have more conceptual context to aid memory , or refresh of memory.

4.5.4 Idea for an exercise

Replace the Linux completely fair scheduler red-black tree implementation with a skip list , and see how your brand of Linux runs after recompiling.

4.6 Treaps

A treap is a two keyed binary tree, that uses a second randomly generated key and the previously discussed tree operation of parent-child rotation to randomly rotate the tree so that overall, a balanced tree is produced. Recall that binary trees work by having all nodes in the left subtree small than a given node, and all nodes in a right subtree greater. Also recall that node rotation does not break this order (some people call it an invariant), but changes the relationship of parent and child, so that if the parent was smaller than a right child, then the parent becomes the left child of the formerly right child. The idea of a tree-heap or treap, is that a binary heap relationship is maintained between parents and child, and that is a parent node has higher priority than its children, which is not the same as the left , right order of keys in a binary tree, and hence a recently inserted leaf node in a binary tree which happens to have a high random priority, can be rotated so it is relatively higher in the tree, having no parent with a lower priority. See the preamble to skip lists about red-black trees on the details of left rotation[2].

A treap is an alternative to both red-black trees, and skip lists, as a self-balancing sorted storage structure.

4.7 Derandomization

[TODO: Deterministic algorithms for Quicksort exist that perform as well as quicksort in the average case and are guaranteed to perform at least that well in all cases. Best of all, no randomization is needed. Also in the discussion should be some perspective on using randomization: some randomized algorithms give you better confidence probabilities than

2 http://en.wikibooks.org/wiki/..%2FLeft%20rotation

the actual hardware itself! (e.g. sunspots can randomly flip bits in hardware, causing failure, which is a risk we take quite often)]

[Main idea: Look at all blocks of 5 elements, and pick the median (O(1) to pick), put all medians into an array (O(n)), recursively pick the medians of that array, repeat until you have < 5 elements in the array. This recursive median constructing of every five elements takes time T(n)=T(n/5) + O(n), which by the master theorem is O(n). Thus, in O(n) we can find the right pivot. Need to show that this pivot is sufficiently good so that we're still O(n log n) no matter what the input is. This version of quicksort doesn't need rand, and it never performs poorly. Still need to show that element picked out is sufficiently good for a pivot.]

4.8 Exercises

1. Write a **find-min** function and run it on several different inputs to demonstrate its correctness.