W1: RAM vs EM Algorithms

- **Due:** Sunday Feb 18
- **Summary:** 3 questions and one challenge question, for a total of 12 points.
- **Submission:** https://autolab.cse.buffalo.edu/courses/cse410-s24/assessments/W1-Binary

Submission

Only PDF-formatted files will be accepted by autolab.

- You may write out your answers by hand and scan them; Numerous apps exist for phones to 'scan' in written documents as a PDF.
- You may typeset your answers in LaTeX, Typst, or a similar tool.

Note that the instructor must be able to read your answer. Submissions that are unintelligible will receive no points.

Binary Search

Recall the basic binary search algorithm.

```
fun binary_search(target: u32, data: Vec<u32>) -> usize
  { return binary_search(target, data, 0, data.len()); }
fun binary_search(target: u32, data: Vec<u32>, start: usize, end: usize) -> usize
{
  if(start >= end-1) { return start }

  let mid = (start - end) / 2 + start;

  if( data[mid] == target )    { return mid; }
  else if( data[mid] < target ){ return binary_search(target, data, mid, end); }
  else                         { return binary_search(target, data, start, mid); }
}
```

Question 1: Binary Search in RAM [4pt]

Let $T(N)$ be the runtime of the binary search algorithm given above where $N =$ `data.len()`:

1. Set up the recurrence relation for $T(N)$ (i.e., define T(N) by cases, in terms of itself).
2. Set up the base and recursive cases for a proof by induction that $T(N) = O(\log_2(N))$
3. Complete the proof by recursion that $T(N) = O(\log_2(N))$

Answer

$$T(N) = \begin{cases} O(1) \text{ if } N{=}1 \text{ or record found} \\ O(1){+}T\left(\frac{N}{2}\right) \text{ otherwise} \end{cases}$$

—

Inductive Hypothesis: $T(N) = O(\log_2(N))$

Base Case: $T(2) = O(\log_2(2))$

(we start with N = 2 because $\log_2(1) = 0$)

Recursive Case: , then $T(N) = O\left(\log_2\left(\frac{N}{2}\right)\right)$
- Precondition: $T\left(\frac{N}{2}\right) = O\left(\log_2\left(\frac{N}{2}\right)\right)$
- Proof Goal: $T(N) = O(\log_2(N))$

—

Is it the case that... $T(2) = O(\log_2(2))$ $\exists c > 0 : T(2) \leq c \cdot \log_2(2)$
$\exists c > 0 : T(2) \leq c \cdot 1$
$\exists c > 0 : 1 + T(1) \leq c \cdot 1$
$\exists c > 0 : 1 + 1 \leq c \cdot 1$
$\exists c > 0 : 2 \leq c$

This statement is true for any $c \geq 2$, so the initial statement must be true.

—

Is it the case that... $T(N) = O(\log_2(N))$
$\exists c > 0 : T(N) = c \cdot \log_2(N)$
$\exists c > 0 : 1 + T\left(\frac{N}{2}\right) = c \cdot \log_2(N)$

Given the precondition, we can replace $T\left(\frac{N}{2}\right) = c \cdot \log_2\left(\frac{N}{2}\right)$
$\exists c > 0 : 1 + c \cdot \log_2\left(\frac{N}{2}\right) = c \cdot \log_2(N)$
$\exists c > 0 : 1 + c \cdot \log_2(N) - \log_2(2) = c \cdot \log_2(N)$
$\exists c > 0 : 1 + c \cdot \log_2(N) - 1 = c \cdot \log_2(N)$
$\exists c > 0 : c \cdot \log_2(N) = c \cdot \log_2(N)$

This statement is true for any value of c, so the initial statement must be true.

—

The proof holds for any $c \geq 2$ and $N \geq 2$

Question 2: Binary Search in EM [4pt]

Assume that:
- `data` is initially stored in external memory (i.e., on disk), as it is in P1.
- Each disk page stores P `u32` values.

Let $I(N)$ be the number of page reads (i.e., the IO Complexity) of the binary search algorithm given above, where N is defined as above.
1. Set up the recurrence relation for $I(N)$.
2. Set up the base and recursive cases for a proof by induction that $I(N) = O(\log_2(N))$

3. Complete the proof by recursion that $I(N) = O(\log_2(N))$

> **Answer**
>
> In the worst case, `data[...]` represents one disk read. The recurrence relation is:
>
> $$I(N) = \begin{cases} O(1) \text{ if } N=1 \text{ or record found} \\ O(1)+I\left(\frac{N}{2}\right) \text{ otherwise} \end{cases}$$
>
> Note that this is exactly the same recurrence relation as part 1. The rest of the setup is identical. Since the goal is to prove an upper bound, proving it for the worst case is sufficient to prove it for a better case (e.g., with caching).
>
> If we cache pages (of size P), the last $O(\log_2(P)) = O(1)$ reads will go to the same page, changing the recurrence relation changes only slightly:
>
> $$I(N) = \begin{cases} O(1) \text{ if } N<\log_2(P) \text{ or record found} \\ O(1)+I\left(\frac{N}{2}\right) \text{ otherwise} \end{cases}$$
>
> This is a legitimate approach; the proof differs only in the use of $\log_2(P)$ as a base case.

ISAM Index

Remember the ISAM index structure we discussed in class? For $N =$ `data.len()` records, and P `u32` values per page, the index is a tree built as follows:

- The 1st level contains P `u32` values on 1 page, taken at uniform intervals from `data`
- The 2nd level contains P^2 `u32` values on P pages taken at uniform intervals from `data`
- …
- The ith level contains P^i `u32` values on P^{i-1} pages, taken at uniform intervals from `data`
- …
- The last level contains all of `data`.

To find a value (let's call this `isam_find`):

1. We do a binary search on the 1st level page. Say the value is between the i and $i + 1$th elements on the 1st level page (or simply greater than the ith element if $i = P - 1$).
2. We do a binary search on the ith page of the 2nd level. Say the value is between the j and $j + 1$th elements on the ith 2nd level page (or greater than the jth element if $j = P - 1$)
3. Repeat the process, descending levels until we identify the specific page of data.

If you prefer code, this algorithm is summarized as follows:

```
fun isam_find(target: u32, data: ISAM) -> Option<usize>
  { isam_find(target, data, 0, 0); }
fun isam_find(target: u32, data: ISAM, level: u32, page: usize) -> Option<usize>
{
  let current_page: Vec<u32> = data.get_page(level, page);
  let position = binary_search(target, current_page);
  if(level >= data.depth())
  {
    return page * data.page_size() + position;
  }
  else
  {
    return isam_find(target, data, level+1, page * data.page_size() + position)
```

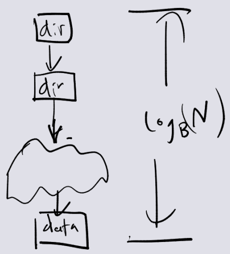
```
  }
}
```

Question 3: ISAM Index in EM [4pt]

Assume that you have an ISAM index structure, as defined above, stored on disk. Let $I_{\text{ISAM}}(N, P)$ be the number of page reads (i.e., the IO Complexity) of the `isam_find` algorithm defined above.

1. Draw the recurrence diagram[1]
2. Use the recurrence diagram to make a guess about the asymptotic bound on $I_{\text{ISAM}}(N, P)$.
3. Set up the recurrence relation for $I_{\text{ISAM}}(N, P)$ given the bound you guessed above.
4. Complete the proof by recursion for the bound you guessed on $I_{\text{ISAM}}(N, P)$.

Answer



Inductive Hypothesis: $I_{\text{ISAM}(N,P)} = \log_{B(N)}$

—

$$I_{\text{ISAM}(N,P)} = \begin{cases} 1 \text{ if } N \leq P \\ 1 + I_{\text{ISAM}\left(\frac{N}{2}, P\right)} \text{ otherwise} \end{cases}$$

—

The proof is identical to part 1, excepting that the base case is $N = P$

Challenge Question [no points]

Assume you have an on-disk array of records **in sorted order**. What is the IO complexity of building an ISAM index, and what is an algorithm that achieves this bound.

[1] e.g., see https://cse.buffalo.edu/courses/cse250/2023-fa/slides/lec12-c.pdf, slide 5

Answer

The following simplified algorithm achieves the complexity bound. An algorithm with a better constant factor exists, but is less concise.

Assume we have the size of the array to start. If we don't, it can be obtained in O(N) IOs.

Maintain $\log_{P(N)}$ in-memory buffered directory pages, one for each level of the tree.

Scan through each page of the on-disk array in-order. Append the first key on the page to the last directory page in the buffer. If the directory page fills up:

1. Write the directory page to disk.
2. Clear the buffered directory page.
3. Direct the next key write one level up in the tree.

```
fn build_isam(data: FileArray, index: ISAM)
{
  let mut buffer = vec![DirectoryPage::alloc();
ceil(log_2(data.len()))]
  let mut level = buffer.len()-1;
  for page in data
  {
    buffer[level].append(page[0])
    if buffer[level].is_full() {
      index.append_directory_page(buffer[level], level)
      level -= 1
    } else {
      level = buffer.len()-1;
    }
  }
}
```

This is an example of a read-once algorithm. Each page is read exactly once, and every page of the ISAM index (of which there are $O(N)$) is written exactly once.