Chapter 1

Induction and Asymptotic Notations.

Computer science differs from other scientific disciplines in that it focuses not on solving or making discoveries, but on questioning how good is our current understanding. The fact that one has successfully come up with an idea for a certain problem immediately raises the question of optimality. At the most basic level, we would like to answer what is the 'best' program that exists for a particular problem. To do so, we must have a notation that allows us to determine if an algorithm is indeed solving the task, quantify its performance, and compare it to other algorithms. In this chapter, we introduce this basic notation. The chapter is divided into two main parts: the first is about induction, a mathematical technique for proving claims, and the second presents asymptotic notation, which we use to describe the behavior of algorithms over large inputs.

Note 1: text for right-hand side of pages, it is set justified.

1.1 Induction.

Suppose that a teacher, who is standing in front of his class, is willing to prove that he can reach the door at the corner. One obvious way to do so is to actually reach the door; that is, move physically to it and declare success. For small classes containing a small number of students, this protocol might even be efficient, lasting less than several seconds. But what if the class is really big, maybe the length and width of a football stadium? In that case, proving by doing might take time. So the obvious question to ask is, what else can we do? Is there a more efficient way to prove this?

Indeed, there is. Instead of proving that he can reach the door, he can prove that while he do not stand next to the door, nothing can stop him from keeping moving forward. If that is indeed the case, then it's clear that not reaching the door in the end would be a contradiction to being just one step away from it (why?), which, in turn, would also contradict being two steps away from it. Repeating this argument leads to a contradiction for the fact that the teacher was in the classroom at the beginning.

What is induction?

1. A mathematical proof technique. It is essentially used to prove that a property $P(n)$ holds for every natural number n.
2. The method of induction requires two cases to be proved:
   (a) The first case, called the base case, proves that the property holds for the first element.
   (b) The second case, called the induction step, proves that if the property holds for one natural number, then it holds for the next natural number.
3. The domino metaphor.

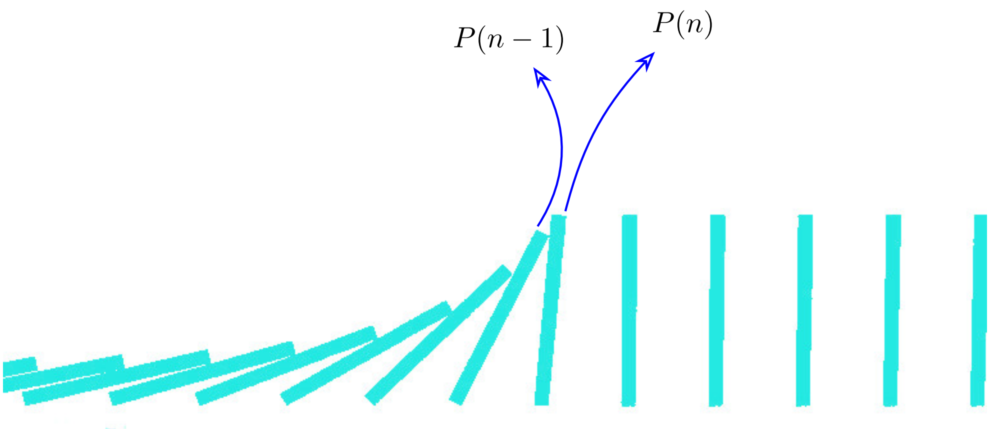


Figure 1.1: domino line falling.

1.1.1 Two Types of Induction Strategies.

We keep in mind two types of induction strategies: the first is Weak Induction, which refers to proofs that, in their step stage, only require assuming the correctness of the previous step, i.e. $P(n-1)$, in order to prove correctness for $P(n)$. This differs from the Strong Induction strategy, for which the step stage assumes the correctness of the claim for any value less than n, namely assuming that all $P(1), P(2), ..., P(n-1)$ are correct.

Example 1.1.3 and Example 1.1.2 demonstrate the use of weak induction to prove the formulas for arithmetic and geometric sums. Example 1.1.1 uses strong induction to determine the number of splits required to separate a chocolate bar.

Example 1.1.1 (Weak induction). Prove that $\forall n \in \mathbb{N}$:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Proof. By induction.

1. Base. For $n = 1$, $\sum_{i=0}^{1} 1 = 1 = \frac{(1+1)\cdot 1}{2}$.
2. Assumption. Assume that the claim holds for n.

3. Step.

$$\sum_{i=0}^{n+1} i = \left(\sum_{i=0}^{n} i\right) + n + 1 = \frac{n(n+1)}{2} + n + 1$$
$$= \frac{n(n+1) + 2 \cdot (n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

□

Example 1.1.2 (Weak induction.). Let $q \in \mathbb{R}^+/\{1\}$, consider the geometric series $1, q, q^2, q^3 \ldots . q^k$. Prove that the sum of the first k elements is

$$1 + q + q^2 + \ldots + q^{k-1} + q^k = \frac{q^{k+1} - 1}{q - 1}$$

Proof. By induction.

1. Base. For $n = 1$, we get $\frac{q^{k+1}-1}{q-1} = \frac{q-1}{q-1} = 1$.
2. Assumption. Assume that the claim holds for an integer k.
3. Step.

$$1 + q + q^2 + \ldots + q^{k-1} + q^k + q^{k+1} = \frac{q^k - 1}{q - 1} + q^{k+1}$$
$$= \frac{q^{k+1} - 1 + q^{k+1}(q-1)}{q-1}$$
$$= \frac{q^{k+1} - 1 + q^{k+2} - q^{k+1}}{q-1}$$
$$= \frac{q^{k+2} - 1}{q-1}$$

□

Example 1.1.3 (Strong induction). Let there be a chocolate bar that consists of n square chocolate blocks. Then it takes exactly $n - 1$ snaps to separate it into the n squares no matter how we split it.

Proof. By strong induction.

1. Base. For $n = 1$, it is clear that we need 0 snaps.
2. Assumption. Assume correctness for **every** $m < n$.
3. Step. We have in our hand the given chocolate bar with n square chocolate blocks. Then we may snap it anywhere we like, to get two new chocolate bars: one with some $k \in [n]$ chocolate blocks and one with $n - k$ chocolate blocks. From the induction assumption, we know that it takes $k - 1$ snaps to separate the first bar, and $n - k - 1$ snaps for the second one. And to sum them up, we got exactly

$$(k-1) + (n-k-1) + 1 = n - 1$$

snaps.

□

1.2 Asymptotic Notations.

Definition 1.2.1. Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f = O(g)$ if there exists $N \in \mathbb{N}$ and $c > 0$ such that for all $n \geq N$ we have $f(n) \leq c \cdot g(n)$.

Example 1.2.1. For example, if $f(n) = n + 10$ and $g(n) = n^2$, then $f = O(g)$ (Draw the graphs) for $n \geq 5$: $f(n) = n + 10 \leq n + 2n = 3n \leq n \cdot n = n^2$

Example 1.2.2. Also if $f(n) = 5n$ and $g(n) = n^2$, then $f(n) = O(g(n))$

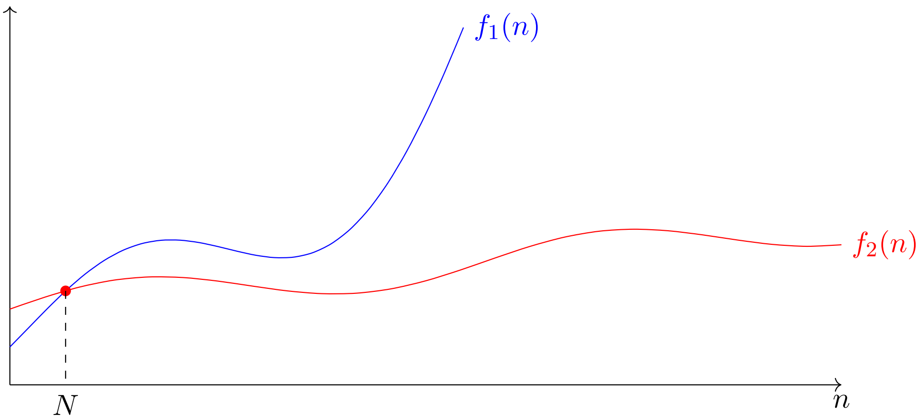


Figure 1.2: Here $f_1(x) \sim x^2 + \sin(x)$, and $f_2 \sim x + \sin(x)$, both defined up to additive and multiplicative terms. So, $f_1 = \Omega(f_2)$ and $f_2 = O(f_1)$.

Definition 1.2.2. Let $f, g : \mathbb{N} \to \mathbb{R}$ We say that $f = \Omega(g)$ if $g = O(f)$, equivalently, there exist $N \in \mathbb{N}$ and $c > 0$ s.t. $\forall n \geq N \Rightarrow cg(n) \leq f(n)$

Example 1.2.3. For example, if $f(n) = n + 10$ and $g(n) = n^2$, then $g = \Omega(f)$

Definition 1.2.3. Let $f, g : \mathbb{N} \to \mathbb{R}$, We say that $f = \Omega(g)$ if: there exist $N \in \mathbb{N}$ and $\exists c > 0$ s.t $\forall n \geq N \Rightarrow f(n) \geq c \cdot g(n)$.

Definition 1.2.4. Let $f, g : \mathbb{N} \to \mathbb{R}$, We say that $f = \Theta(g)$ if: $f = O(g)$ and $f = \Omega(g)$ That is, we say that $f = \Theta(g)$ if: $\exists N \in \mathbb{N}, \exists c_1, c_2 > 0$ s.t. $\forall n \geq N$ $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Example 1.2.4. For every $f : \mathbb{N} \to \mathbb{R}$, $f(n) = \Theta(f(n))$

Example 1.2.5. If $p(n) = n^5$ and $q(n) = 0.5n^5 + n$, then $p(n) = \Theta(q(n))$

But why is this example true? This next Lemma helps for intuition:

Lemma 1.2.1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = O(g(n))$

Proof. Assume that $l = \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$. Then for some $N \in \mathbb{N}$ we have that for all $n \geq N$: $\frac{f(n)}{g(n)} < l + 1 \Rightarrow f(n) < (l + 1)g(n)$ Which is exactly what we wanted. □

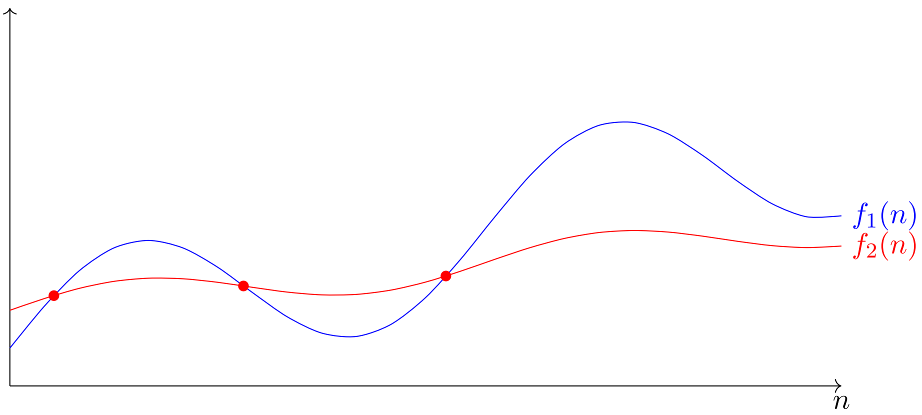


Figure 1.3: Here both $f_1(x)$ and $f_2(x)$ behave like $\sim x + \sin(x)$ up to additive and multiplicative terms. So, $f_1 = \Theta(f_2)$.

1.3 Examples with proofs.

Claim 1.3.1. $n = O(2^n)$

(This must seem very silly, but even though we have a strong feeling it's true, we still need to learn how to PROVE it)

Proof. We will prove by induction that $\forall n \geq 1$, $2^n \geq n$, and that will suffice.

1. Base. $n = 1$, so it is clear that: $n = 1 < 2 = 2^n$
2. Assumption. Assume that $n < 2^n$ for some n.
3. Step. We will prove for $n + 1$. It holds that:

$$n + 1 < 2^n + 1 < 2^n + 2^n = 2^{n+1}$$

□

Claim 1.3.2. *Let $p(n)$ be a polynomial of degree d and let $q(n)$ be a polynomial of degree k. Then:*

1. *$d \leq k \Rightarrow p(n) = O(q(n))$ (set upper bound over the quotient)*
2. *$d \geq k \Rightarrow p(n) = \Omega(q(n))$ (an exercise)*
3. *$d = k \Rightarrow p(n) = \Theta(g(n))$ (an exercise)*

Proof. Proof (Of 1) First, let's write down $p(n), g(n)$ explicitly:

$$p(n) = \sum_{i=0}^{d} \alpha_i n^i, \ g(n) = \sum_{j=0}^{k} \beta_j n^j$$

Now let's manipulate their quotient:

$$\frac{p(n)}{q(n)} = \frac{\sum_{i=0}^{d} \alpha_i n^i}{\sum_{j=0}^{k} \beta_j n^j} = \frac{\sum_{i=0}^{d} \alpha_i n^i}{\sum_{j=0}^{k} \beta_j n^j} \cdot \frac{n^{k-1}}{n^{k-1}} = \frac{\sum_{i=0}^{d} \alpha_i n^{i-k+1}}{\sum_{j=0}^{k} \beta_j n^{j-k+1}} \leq$$

$$\leq \frac{\sum_{i=0}^{d} \alpha_i}{\beta_k} < \infty$$

And now we can use the lemma that we have proved earlier. □

1.4 Logarithmic Rules.

Just a quick reminder of logarithmic rules:

1. $log_a x \cdot y = log_a x + log_a y$
2. $log_a \frac{x}{y} = log_a x - log_a y$
3. $log_a x^m = m \cdot log_a x$
4. Change of basis: $\frac{log_a x}{log_a y} = log_y x$

And so we get that:

Remark 1.4.1. For every $x, a, b \in \mathbb{R}$, we have that $log_a x = \Theta(log_b x)$

Example 1.4.1. Let $f(n)$ be defined as:

$$f(n) = \begin{cases} f\left(\lfloor \frac{n}{2} \rfloor\right) + 1 & \text{for } n > 1 \\ 5 & \text{else} \end{cases}$$

Let's find an asymptotic upper bound for $f(n)$. let's guess $f(n) = O(\log(n))$.

Proof. We'll prove by strong induction that : $f(n) < c\log(n) - 1$ for $c = 8$ And that will be enough (why? This implies $f(n) = O(\log(n))$).

1. Base. $n = 2$. Clearly, $f(2) = 6 < 8$
2. Assumption. Assume that for every m ¡ n, this claim holds.
3. Step. Then we get:

$$\begin{aligned} f(n) = f\left(\lfloor \frac{n}{2} \rfloor\right) + 1 &\leq c\log\left(\lfloor \frac{n}{2} \rfloor\right) + 1 \\ &\leq c\log(n) - c\log(2) + 1 \leq c\log(n) \quad \text{for } c = 8 \end{aligned}$$

□

Chapter 2

Introduction to Algorithms.

2.1 Peaks-Finding.

Example 2.1.1 (Leading Example.). Consider an n-length array A such that $A_1, A_2, ..., A_n \in \mathbb{R}$. We will say that A_j is a peak (local minimum) if he's greater than his neighbors. Namely, $A_i \geq A_{i\pm1}$ if $i \pm 1 \in [n]$. Whenever $i \pm 1$ is not in the range $[n]$, we will define the inequality $A_i \geq A_{i\pm1}$ to hold trivially. For example, for $n = 1$, $A_1 = A_n$ is always a peak. Write an algorithm that, given A, returns the position of an arbitrary peak.

Example 2.1.2. Warming up. How many peaks do the following arrays contain?

1. $A[i] = 1 \; \forall i \in [n]$
2. $A[i] = \begin{cases} i & i < n/2 \\ n/2 - i & \text{else} \end{cases}$
3. $A[i] = i \; \forall i \in [n]$

2.2 Naive solution.

To better understand the problem, let's first examine a simple solution before proposing a more intriguing one. Consider the algorithm examining each of the items A_i one by one.

Result: returns a peak of $A_1...A_n \in \mathbb{R}^n$
for $i \in [n]$ **do**
  if A_i *is a peak* **then**
    return i
  end
end

Algorithm 1: naive peak-find alg.

Correctness. We will say that an algorithm is correct, with respect to a given task, if it computes the task for any input. Let's prove that the above algorithm is doing the job.

Proof. Assume towards contradiction that there exists an n-length array A such that the algorithm peak-find fails to find one of its peaks, in particular, the Alg. either returns $j' \in [n]$ such that $A_{j'}$ is not a peak or does not return at all (never reach line (3)). Let's handle first the case in which returning indeed occurred. Denote by j the first position of a peak in A, and note that if the algorithm gets to line (2) in the jth iteration then either it returns j or A_j is not a peak. Hence it must hold that $j' < j$. But satisfaction of the condition on line (2) can happen only if $A_{j'}$ is a peak, which contradicts the minimality of j. In the case that no position has been returned, it follows that the algorithm didn't return in any of the first j iterations and gets to iteration number $j+1$, which means that the condition on line (2) was not satisfied in contradiction to the fact that A_j is a peak. □

Running Time. Question, How would you compare the performance of two different algorithms? What will be the running time of the naive peak-find algorithm? On the lecture you will see a well-defined way to treat such questions, but for the sake of getting the general picture, let's assume that we pay for any comparison a quanta of processing time, and in overall, checking if an item in a given position is a peak, cost at most $c \in \mathbb{N}$ time, a constant independent on n.

Question, In the worst case scenario, how many local checks does peak-finding do? For the third example in Example 2.1.2, the naive algorithm will have to check each item, so the running time adds up to at most $c \cdot n$.

2.3 Naive alg. recursive version.

Now, we will show a recursive version of the navie peak-find algorithm for demonstrating how correctness can be proved by induction.

```
Result: returns a peak of A_1...A_n ∈ R^n
if A_1 ≥ A[2] or n = 1 then
    return 1
end
return 1 + peak-find(A_2, ..A_n)
```

Algorithm 2: naive recursive peak-find alg.

Claim 2.3.1. *Let $A = A_1, \ldots, A_n$ be an array, and $A' = A_2, A_3, \ldots, A_n$ be the $n-1$ length array obtained by taking all of A's items except the first. If $A_1 \leq A_2$, then any peak of A' is also a peak of A.*

Proof. Let A'_j be a peak of A'. Split into cases upon on the value of j. If $n-1 > j > 1$, then $A'_j \geq A'_{j\pm1}$, but for any $j \in [2, n-2]$ we have $A'_j = A_{j+1}$ and therefore $A_{j+1} \geq A_{j+1\pm1} \Rightarrow A_{j+1}$ is a peak in A. If $j' = 1$, then $A'_1 > A'_2 \Rightarrow A_2 \geq A_3$ and by combining the assumption that $A_1 \leq A_2$ we have that $A_2 \geq A_1, A_3$. So $A_2 = A'_1$ is also a peak. The last case $j = n-1$ is left as an exercise. □

One can prove a much more general claim by following almost the same argument presented above.

Claim 2.3.2. *Let $A = A_1, \ldots, A_n$ be an array, and $A' = A_{j+1}, A_{j+2}, \ldots, A_n$ be the $n-j$ length array. If $A_j \leq A_{j+1}$, then any peak of A' is also a peak of A.*

We are ready to prove the correctness of the recursive version by induction using Claim 2.3.1.

1. Base, single element array. Trivial.

2. Assumption, Assume that for any m-length array, such that $m < n$ the alg returns a peak.

3. Step, consider an array A of length n. If A_1 is a peak, then the algorithm answers affirmatively on the first check, returning 1 and we are done. If not, namely $A_1 < A_2$, then by using Claim 2.3.1 we have that any peak of $A' = A_2, A_3, \dots, A_n$ is also a peak of A. The length of A' is $n - 1 < n$. Thus, by the induction assumption, the algorithm succeeds in returning on A' a peak which is also a peak of A.

2.4 An attempt for sophisticated solution.

We saw that we can find an arbitrary peak at $c \cdot n$ time, which raises the question, can we do better? Do we really have to touch all the elements to find a local maxima? Next, we will see two attempts to catch a peak at logarithmic cost. The first attempt fails to achieve correctness, but analyzing exactly why will guide us on how to come up with both an efficient and correct algorithm.

Result: returns a peak of $A_1 ... A_n \in \mathbb{R}^n$
$i \leftarrow \lceil n/2 \rceil$
if A_i *is a peak* **then**
    return i
end
else
    return $i - 1+$ find-peak$(A_i, A_{i+1} .. A_n)$
end

Algorithm 3: fail attempt for more sophisticated alg.

Let's try to 'prove' it.

1. Base, single element array. Trivial.

2. Assumption, Assume that for any m-length array, such that $m < n$ the alg returns a peak.

3. Step. If $A_{n/2}$ is a peak, we're done. What happens if it isn't? Is it still true that any peak of $A_i, A_{i+1}, \dots, A_n$ is also a peak of A? Consider, for example, $A[i] = n - i$.

2.5 Sophisticated solution.

The example above points to the fact that we would like to have a similar claim to Claim 2.3.2 that relates the peaks of the split array to the original one. Let's prove correction by induction.

Result: returns a peak of $A_1 ... A_n \in \mathbb{R}^n$
$i \leftarrow \lceil n/2 \rceil$
if A_i *is a peak* **then**
  return i
end
else if $A_{i-1} \leq A_i$ **then**
  return $i+$ find-peak$(A_{i+1} .. A_n)$
end
else
  return find-peak$(A_1, A_2, A_3 .. A_{i-1})$
end

Algorithm 4: sophisticated alg.

Proof.

1. Base, single element array. Trivial.
2. Assumption, Assume that for any m-length array, such that $m < n$ the alg returns a peak.
3. Step, Consider an array A of length n. If $A_{\lceil n/2 \rceil}$ is a peak, then the algorithm answers affirmatively on the first check, returning $\lceil n/2 \rceil$ and we are done. If not, then either $A_{\lceil n/2 \rceil} < A_{\lceil n/2 \rceil - 1}$ or $A_{\lceil n/2 \rceil} < A_{\lceil n/2 \rceil + 1}$. We have already handled the first case, that is, using Claim 2.3.2 we have that any peak of $A' = A_{\lceil n/2 \rceil + 1}, A_{\lceil n/2 \rceil + 2}, \ldots, A_n$ is also a peak of A. The length of A' is $n/2 < n$. So by the induction assumption, in the case where $A_{\lceil n/2 \rceil} < A_{\lceil n/2 \rceil - 1}$ the algorithm returns a peak. In the other case, we have $A_{\lceil n/2 \rceil} < A_{\lceil n/2 \rceil + 1}$ (otherwise $A_{\lceil n/2 \rceil}$ would be a peak). We leave finishing the proof as an exercise.

□

What's the running time? Denote by $T(n)$ an upper bound on the running time. We claim that $T(n) \leq c_1 \log(n) - c_2$, let's prove it by induction.

Proof.

1. Base. For the base case, $n \leq 3$ we get that $c_1 \log(1) + c_2 = c_2$ on the other hand only a single check made by the algorithm, so indeed the base case holds (Choosing $c_2 \geq$ the cost of a single check).
2. Induction Assumption. Assume that for any $m < n$, the algorithm runs in at most $c_1 \log(m) + c_2$ time.
3. Step. Notice that in the worst case, $\lceil n/2 \rceil$ is not a peak, and the algorithm calls itself recursively immediately after paying c_2 in the first check. Hence:

$$\begin{aligned} T(n) &\leq c_2 + T(n/2) \leq c_2 + c_1 \log(\lceil n/2 \rceil) + c_2 \\ &= c_1 - c_1 + c_2 + c_1 \log(\lceil n/2 \rceil) + c_2 \\ &= c_1 \log(2) + c \log(\lceil n/2 \rceil) + 2c_2 - c_1 \\ &= c_1 \log(2(\lceil n/2 \rceil)) + 2c_2 - c_1 \end{aligned}$$

So, choosing $c_1 > c_2$ gives $2c_2 - c_1 < c_2$ and therefore:

$$T(n) \leq c_1 \log(n) + c_2$$

□

Chapter 3

DAST. Exercise 0.

3.1 Peaks-Finding.

Recall the logarithmic-time version of find-peak. Prove its correctness. You can continue directly from the point we have reached to in the recitation.

Result: returns a peak of $A_1 ... A_n \in \mathbb{R}$
$i \leftarrow \lceil n/2 \rceil$
if A_i *is a peak* **then**
  return i
end
else if $A_{i-1} \leq A_i$ **then**
  return $i+$ find-peak$(A_{i+1}..A_n)$
end
else
  return find-peak$(A_1, A_2, A_3..A_{i-1})$
end

Algorithm 5: sophisticated alg.

3.2 Peaks-Finding on the Cycle.

Consider the following variation of the peaks-finding problem. We will define the peaks again to be the local maximas of given array A. However instead of thinking on position $1, 2..n$ as coordinates on the line we will think about them as cordinates on the cycle. Namely A_1 is the right neighbor of A_n.

Show a series of arrays $\{A_n\}_{n=3}^{\infty}$ for which the recursive version of peak-find always fails. Explain in one line why the naive algorithm is still correct (You don't have to provide a formal correctness proof).

3.3 k-range peaks.

In this section we define k-range peak to be all the items satisfiey:

$$A_i = \max_{|i-j| \leq k} \{A_j\}$$

Write an algorithm that find a k-range peak in at most $k\log(n)$ up to constant factor indepandet on k and n. Prove correctness, and bound the running time.

3.4 2D-peaks.

Let A be a $n \times n$ matrix. We will say that A_{ij} is a peak if:

$$A_{ij} \geq A_{i-1,j}, A_{i+1,j}, A_{i,j-1}, A_{i,j+1}$$

And again if the indices exceed the matrix frame, we define the inequality as satisfied trivially (For example if $A_{1,1} \geq A_{1,2}, A_{2,1}$ then $A_{1,1}$ is a peak).

Write an algorithm that find a peak in A. Solutions that runs in at most $cn\log n$ are sufficient. Prove correctness, and bound the running time.

Chapter 4

Correctness - Recitation 2

Proving algorithms correctness is necessary to guarantee that our code computes its goal for every given input. In that recitation, we will examine several algorithms, analyze theirs running time and memory consumption, and prove they are correct.

Example 4.0.1 (Leading Example.). Consider n numbers $a_1, a_2, ..., a_n \in \mathbb{R}$. Given set Q of $|Q|$ queries, such each query $q \in Q$ is a tuple $(i, j) \in [n] \times [n]$. Write an algorithm that calculates the $\max_{i \leq k \leq j} a_k$.

4.1 Correctness And Loop Invariant.

Correctness. We will say that an algorithm $\mathcal{A}$ (an ordered set of operations) computes $f : D_1 \to D_2$ if for every $x \in D_1$ the following equality holds $f(x) = \mathcal{A}(x)$. Sometimes when it's obvious what is the goal function f, we will abbreviate and say that $\mathcal{A}$ is correct.

Examples of functions f might be: file saving, summing numbers, or posting a message in the forum.

Loop Invariant. Loop Invariant is a property that characteristic a loop segment code and satisfy the following conditions:

Note 1: text for right-hand side of pages, it is set justified.

1. Initialization. The property holds (even) before the first iteration of the loop.
2. Conservation. As long as one performs the loop iterations, the property still holds.
3. (optional) Termination. Exiting from the loop carrying information.

Example 4.1.1. Before dealing with the hard problem, let us face the naive algorithm to find the maximum of a given array.

Claim 4.1.1. *Consider the while loop. The property: "for every $j' < j \leq n + 1 \Rightarrow a_{j'} \leq a_i$" is a loop invariant that is associated with it.*

Proof. first, the initialization condition holds, as the at the first iteration $j = 1$ and therefore the property is trivial. Assume by induction, that for every $m < j$ the property is correct, and consider the j-th iteration. If back again to line (5), then it means that $(j - 1) < n$ and $a_{j-1} \leq a_i$. Combining the above with the induction assumption yields that $a_i \geq a_{j-1}, a_{j-2}, ...a_1$. □

```
   Result: returns the maximum of a_1...a_n ∈ ℝ^n
1  Function max( a_1..a_n )
2      for i ∈ [n] do
3          j ← 1
4
5          while j ≤ [n] and a_i ≥ a_j do
6              j ← j + 1
7          end
8
9          if j = n + 1 then
10             return a_i
11         end
12     end
13     return Δ
14 end
```

Algorithm 6: naive maximum alg.

Correctness Proof. Split into cases, First if the algorithm return result at line (9), then due to the loop invariant, combining the fact that $j = n + 1$, it holds that for every $j' \leq n < j \Rightarrow a_i \geq a_{j'}$ i.e a_i is the maximum of $a_1, ...a_n$. The second case, in which the algorithm returns Δ at line number (10) contradicts the fact that n is finite, and left as an exercise. the running time is $O(n^2)$ and the space consumption is $O(n)$.

Loop Invariant In The Cleverer Alg. Consider now the linear time algorithm:

```
Result: returns the maximum of a_1...a_n ∈ ℝ^n
Function max( a_i..a_j)

    let b ← a_1

    for i ∈ [2, n] do
        b ← max (b, a_i)
    end
    return b
end
```

Algorithm 7: maximum alg.

What is the Loop Invariant here? *"at the i-th iteration, $b = \max\{a_1...a_{i-1}\}$"*. The proof is almost identical to the naive case.

4.2 Non-Linear Space Complexity Algorithms.

Sub-Array Maximum. Consider the leading example; It's easy to write an algorithm that answers the queries at a total time of a $O(|Q| \cdot n)$ by answers separately on each query. Can we achieve a better upper bound?

Result: print the $\max\{a_i...a_j\}$ for each query $(i,j) \in Q$

```
Function max( a_i...a_j )
    let A ← M^{n×n}

    for i ∈ [n] do
        A_{i,i} ← a_i
    end

    for k ∈ [1,n] do
        for i ∈ [n] do
            if i + k ≤ n then
                A_{i,i+k} ← max(A_{i,i+k-1}, a_{i+k})
            end
        end
    end

    for q ∈ Q do
        i, j ← q
        print A_{i,j}
    end
end
```

Algorithm 8: Sub-Array. $O(n^2)$ space alg.

Claim. Consider the outer loop at the k-th step. The following is a loop invariant:

$$for\ every\ k' < k,\ s.t\ i + k' \leq n \Rightarrow A_{i,i+k'} = \max\{a_i, a_{i+1}, ..., a_{i+k'}\}$$

Proof. The initialization condition trivially holds, assume by induction that $A_{i,i+k-1} = \max\{a_i...a_{i+k-1}\}$ at beginning of k iteration. By the fact that $\max(x,y,z) = \max(\max(x,y),z)$ we get that

$$\max\{a_1...a_{i+k-1}, a_{i+k}\} = \max\{\max\{a_1...a_{i+k-1}\}, a_{i+k}\} = \max\{A_{i,i+k-1}, a_{i+k}\}$$

And the right term is exactly the value which assigned to $A_{i,i+k}$ in the end of the$k$-th iteration. Thus in the beginning of $k+1$ iteration the property is still conserved.

$O(n \log n)$ **Space Solution.** Example for $O(n\log n + |Q|\log n)$ time and $O(n\log n)$ space algorithm. Instead of storing the whole matrix, we store only logarithmic number of rows.

Result: print the $\max\{a_i ... a_j\}$ for each query $(i, j) \in Q$

Function *max(* $a_i ... a_j$ *)*
  let $A \leftarrow \mathbb{M}^{n \times \log n}$

  for $i \in [n]$ **do**
    $A_{i,1} \leftarrow a_i$
  end

  for $k \in [2, 4, .., 2^m, ..., n]$ **do**
    for $i \in [n]$ **do**
      if $i + k \leq n$ **then**
        $A_{i,k} \leftarrow \max\left(A_{i,\frac{k}{2}}, A_{i+\frac{k}{2},\frac{k}{2}}\right)$
      end
    end
  end

  for $q \in Q$ **do**
    $i, j \leftarrow q$
    decompose $j - i$ into binary representation $2^{t_1} + 2^{t_2} + .. + 2^{t_l}$
    print $\max\{A_{i,2^{t_1}}, A_{i+2^{t_1},2^{t_2}}, ..., A_{i+2^{t_1}+2^{t_2}+..2^{t_{l-1}},2^{t_l}}\}$
  end
end

Algorithm 9: Sub-Array. $O(n \log n)$ space alg.

Chapter 3

Recursive Analysis.

3.1 Bounding recursive functions by hands.

Our primary tool to handle recursive relation is the Master Theorem, which was proved in the lecture. As we would like to have a more solid grasp, let's return on the calculation in the proof over a specific case. Assume that your algorithm analysis has brought the following recursive relation:

Example 3.1.1. $T(n) = \begin{cases} 4T\left(\frac{n}{2}\right) + c \cdot n & \text{for } n > 1 \\ 1 & \text{else} \end{cases}$. Thus, the running time is given by

$$T(n) = 4T\left(\frac{n}{2}\right) + c \cdot n = 4 \cdot 4T\left(\frac{n}{4}\right) + 4c \cdot \frac{n}{2} + c \cdot n = \dots =$$
$$\overbrace{4^h T(1)}^{\text{critical}} + c \cdot n\left(1 + \frac{4}{2} + \left(\frac{4}{2}\right)^2 \dots + \left(\frac{4}{2}\right)^{h-1}\right) = 4^h + c \cdot n \cdot \frac{2^h - 1}{2 - 1}$$

We will call the number of iteration till the stopping condition the recursion height, and we will denote it by h . What should be the recursion height? $2^h = n \Rightarrow h = \log(n)$. So in total we get that the algorithm running time equals $\Theta\left(n^2\right)$.

Question, Why is the term $4^h T(1)$ so critical? Consider the case $T(n) = 4T\left(\frac{n}{2}\right) + c$ .One popular mistake is to forget the final term, which yields a linear solution $\Theta(n)$ (instead of quadric $\Theta(n^2)$).

Example 3.1.2. $T(n) = \begin{cases} 3T\left(\frac{n}{2}\right) + c \cdot n & \text{for } n > 1 \\ 1 & \text{else} \end{cases}$, and then the expanding yields:

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{2}\right) + c \cdot n = 3^2 T\left(\frac{n}{2^2}\right) + \frac{3}{2}cn + c \cdot n \\ &= \overbrace{3^h T(1)}^{\text{critical}} + cn\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \dots + \left(\frac{3}{2}\right)^{h-1}\right) \\ h &= \log_2(n) \Rightarrow T(n) = 3^h T(1) + c \cdot n \cdot \left(\left(\frac{3}{2}\right)^{\log_2 n}\right) / \left(\frac{3}{2} - 1\right) \\ &= \theta\left(3^{\log_2(n)}\right) = \theta\left(n^{\log 3}\right) \end{aligned}$$

where $n^{\log 3} \sim n^{1.58} < n^2$.

3.2 Master Theorem, one Theorem to bound them all.

As you might already notice, the same pattern has been used to bound both algorithms. The master theorem is the result of the recursive expansion. it classifies recursive functions at the form of $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, for positive function $f : \mathbb{N} \to \mathbb{R}^{+}$.

Master Theorem, simple version.

First, Consider the case that $f = n^c$. Let $a \geq 1, b > 1$ and $c \geq 0$. then:

1. if $\frac{a}{b^c} < 1$ then $T(n) = \Theta(n^c)$ (f **wins**).
2. if $\frac{a}{b^c} = 1$ then $T(n) = \Theta(n^c \log_b(n))$.
3. if $\frac{a}{b^c} > 1$ then $T(n) = \Theta\left(n^{\log_b(a)}\right)$ (f **loose**).

Example 3.2.1. $T(n) = 4T\left(\frac{n}{2}\right) + d \cdot n \Rightarrow T(n) = \Theta\left(n^2\right)$ according to case (3). And $T(n) = 3T\left(\frac{n}{2}\right) + d \cdot n \Rightarrow T(n) = \Theta\left(n^{\log_2(3)}\right)$ also due to case (3).

Master Theorem, strong version.

Now, let's generalize the simple version for arbitrary positive f and let $a \geq 1$, $b > 1$.

1. if $f(n) = O\left(n^{\log_b(a)-\varepsilon}\right)$ for some $\varepsilon > 0$ then $T(n) = \theta\left(n^{\log_b(a)}\right)$ (f **loose**).
2. if $f(n) = \Theta\left(n^{\log_b(a)}\right)$ then $T(n) = \Theta\left(n^{\log_b(a)} \log(n)\right)$
3. if there exist $\varepsilon > 0, c < 1$ and $n_0 \in \mathbb{N}$ such that $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ and for every $n > n_0$ $a \cdot f\left(\frac{n}{b}\right) \leq cf(n)$ then $T(n) = \theta(f(n))$ (f **wins**).

Example 3.2.2.

1. $T(n) = T\left(\frac{2n}{3}\right) + 1 \to f(n) = 1 = \Theta\left(n^{\log_{\frac{3}{2}}(1)}\right)$ matches the second case. i.e $T(n) = \Theta\left(n^{\log_{\frac{3}{2}}(1)} \log n\right)$.

2. $T(n) = 3T\left(\frac{n}{4}\right) + n \log n \to f(n) = \Omega\left(n^{\log_4(3)+\varepsilon}\right)$ and notice that $af\left(\frac{n}{b}\right) = \frac{3n}{4} \log\left(\frac{n}{4}\right)$. Thus, it's matching to the third case. $\Rightarrow T(n) = \Theta(n \log n)$.

3. $T(n) = 3T\left(n^{\frac{1}{3}}\right) + \log\log n$. Let $m = \log n \Rightarrow T(n) = T(2^m) = 3T\left(2^{\frac{m}{3}}\right) + \log m$. Denote by $S = S(m) = T(2^m) \to S(m) = 3T\left(2^{\frac{m}{3}}\right) + \log m = 3S\left(\frac{m}{3}\right) + \log m$. And by the fact that $\log m = O\left(m^{\log_3(3)-\varepsilon}\right) \to T(n) = T(2^m) = S(m) = \Theta(m) = \Theta(\log(n))$.

3.3 Recursive trees.

There are still cases which aren't treated by the *Master Theorem*. For example consider the function $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$. Note, that $f = \Omega\left(n^{\log_b(a)}\right) = \Omega(n)$. Yet for every $\varepsilon > 0 \Rightarrow f = n\log n = O\left(n^{1+\varepsilon}\right)$ therefore the third case doesn't hold. How can such cases still be analyzed?

Recursive trees Recipe

1. draw the computation tree, and calculate it's height. in our case, $h = \log n$.
2. calculate the work which done over node at the k-th level, and the number of nodes in each level. in our case, there are 2^k nodes and over each we perform $f(n) = \frac{n}{2^k}\log\left(\frac{n}{2^k}\right)$ operations.
3. sum up the work of the k-th level.
4. finally, the total time is the summation over all the $k \in [h]$ levels.

applying the above, yields

$$\begin{aligned} T(n) &= \sum_{k=1}^{\log n} n \cdot \log\left(\frac{n}{2^k}\right) = n\sum_{k=1}^{\log n}\left(\log n - \log 2^k\right) = n\sum_{k=1}^{\log n}(\log n - k) \\ &= \Theta\left(n\log^2(n)\right) \end{aligned}$$

Example 3.3.1. Consider merge sort variation such that instead of splitting the array into two equals parts it's split them into different size arrays. The first one contains $\frac{n}{10}$ elements while second contains the others $\frac{9n}{10}$ elements.

Result: returns the sorted permutation of $x_1 ... x_n \in \mathbb{R}^n$

if $n \leq 10$ **then**
    return bubble-sort $(x_1 ... x_n)$
end

else
    define $S_l \leftarrow x_1 ... x_{\frac{n}{10}-2}, x_{\frac{n}{10}-1}$
    define $S_r \leftarrow x_{\frac{n}{10}}, x_{\frac{n}{10}+1} ..., x_n$

    $R_l \leftarrow$ non-equal-merge(S_l)
    $R_r \leftarrow$ non-equal-merge(S_r)

    return Merge(R_l, R_r)
end

Algorithm 10: non-equal-merge alg.

Note, that the master theorem achieves an upper bound,

$$\begin{aligned} T(n) &= n + T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) \leq n + 2T\left(\frac{9n}{10}\right) \\ \Rightarrow T(n) &= O\left(n^{\log_{\frac{10}{9}}(2)}\right) \sim O\left(n^6\right) \end{aligned}$$

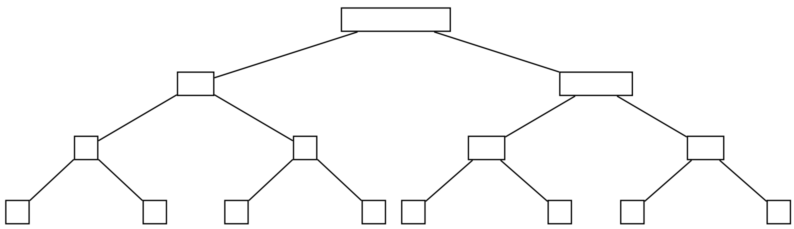


Figure 3.1: The tree matches the recursive calls made by Algorithm 10. Each node presents a rectangle with a length equal to the array given as input to the recursive call. The length of all the elements in a single level is equal to the original array length, thus we have that the linear work in each level sums up to $\Theta(n)$.

Yet, that bound is far from been tight. Let's try to count the operations for each node. Let's try another direction.

Claim 3.3.1. *Let n_i be the size of the subset which is processed at the i-th node. Then for every k:*

$$\sum_{i \in k \text{ level}} n_i \leq n$$

Proof. Assuming otherwise implies that there exist index j such that x_j appear in at least two different nodes in the same level, denote them by u, v. As they both are in the same level, non of them can be ancestor of the other. denote by $m \in \mathbb{N}$ the input size of the sub array which is processed by the the lowest common ancestor of u and v, and by $j' \in [m]$ the position of x_j in that sub array. By the definition of the algorithm it steams that $j' < \frac{m}{10}$ and $j' \geq \frac{m}{10}$. contradiction. The height of the tree is bounded by $\log_{\frac{9}{10}}(n)$. Therefore the total work equals $\Theta\left(n \log n\right)$. Thus, the total running time equals to:

$$T(n) = \sum_{k \in \text{levels}} \sum_{i \in \text{k level}} f\left(n_i\right) = \sum_{k \in \text{levels}} \sum_{i \in \text{k level}} n_i \leq n \log n$$

$\square$

Chapter 3

DAST. Exercise 2.

[COMMENT] Please note that sections 1, 2, and 4 are optional and do not need to be submitted. However, please ensure that you feel capable of solving them. The problems in the following chapter, 7, 8, and 9, are also optional but are more challenging. It is completely acceptable if you are not able to manage them at this stage of the course.

3.1 Small o, ω, True or False? (Not a Mandatory).

Let $f, g, h, k : \mathbb{N} \to \mathbb{R}^+$. Either prove or provide a counterexample:

1. $f = o(g) \Leftrightarrow g = \omega(f)$.
2. $f = O(g) \Rightarrow f = o(g)$.
3. If $h = o(f)$ and $k = o(g)$ then $h/k = o(f/g)$.

3.2 Logaritmic Arithmetic Recap. (Not a Mandatory).

Prove that for any $x, y > 0$:

1. we have $x > y$ if and only if $\log x > \log y$.
2. $x^{\log_y n} = n^{\log_y x}$ For any $n > 0$.
3. $\log(x \pm y) = \log(x) + \log(1 \pm y/x)$.

3.3 Merge Sort Correction.

1. Consider the Merge subroutine used in the Merge-Sort algorithm. Let A and B be the given input arrays, and let C be the output array returned by the subroutine. State, in your own words, a claim referring to the structure of C at the ith iteration. Prove it.
2. Prove the correctness of Merge using your claim.

3.4 Master Theorem. (Not a Mandatory).

Bound the following recursive functions. In each case, assume that $T(1) = T(0) = 1$.

1. $T(n) = 7T(n/3) + n^3$
2. $T(n) = T(\sqrt{n}) + 2^n$

3.5 β-Root Master Theorem Varition.

Consider the following recursice ralation:

$$T(n) = \begin{cases} \alpha T(n^{\frac{1}{\beta}}) + \log^{\gamma}(n) & \text{for } n > 1 \\ 1 & \text{else} \end{cases}$$

When $\alpha, \beta > 1$ and $\gamma \geq 0$. Bound T asymptotically tight, namely find $g : \mathbb{N} \to \mathbb{R}^+$ such that $T = \Theta(g)$. Notice that the solution depends on the relation between α, β, γ.

3.6 Recursive Trees.

Let $\alpha \in (0, 1)$, $l \geq 2$ and let $T : \mathbb{N} \to \mathbb{R}^+$ definied as follow:

$$T(n) = \begin{cases} n^l + T(\alpha n) + T((1-\alpha)n) & \text{for } n > 1 \\ 1 & \text{else} \end{cases}$$

Bound T asymptotically tight.

Extra Questions. (Not a Mandatory).

3.7 T_1, T_2-Running time. (Not a Mandatory).

Let $a, b > 0$ and $T_1, T_2 : \mathbb{N} \to \mathbb{R}^+$ be functions satisfying the equations:

$$T_1(n) = aT_1\left(\frac{n}{2}\right) + bT_2\left(\frac{n}{2}\right) + n^3$$
$$T_2(n) = bT_1\left(\frac{n}{2}\right) + aT_2\left(\frac{n}{2}\right) + n^2$$

for any $n > 1$, and equal to 1 otherwise. Bound T_1, T_2 asymptotically tight. **Hint:** Use your linear algebra arsenal.

3.8 Tensor-calculus-program running time analysis. (Not a Mandatory).

Let $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, ..., \mathcal{A}_m$ be algorithms, where each of their source codes is at most N bits long. These algorithms are restricted to the following format: any operation they perform is either iterating over a k-nested loop and performing a finite amount of work in each iteration, or calling one of the other algorithms on an input of length $n/2$. Additionally, it is assumed that any $\mathcal{A}_i$ can be called by $\mathcal{A}_j$ a finite number of times, independent of N and m. To clarify, calls to other algorithms cannot be made inside one of the for loops.

Write an algortihm that given the sources of $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, ..., \mathcal{A}_m$ bound assymptoticly tight the running time of $\mathcal{A}_1$ at time $\Theta\left(N \cdot m + m^3\right)$.

3.9 2D-Peaks in linear time. (Not a Mandatory).

Write an algorithm that, given a matrix $M \in \mathbb{R}^{n \times n}$, returns a local peak in linear time. Prove correctness.

Chapter 5

Recap Recitation.

5.1 More Sorting, More Correctness.

Until now, all the algorithms that we have seen were, in some sense, intuitive. We could describe in words, step by step, exactly what the algorithm does. For example, bubble and heapsort both bubble up the greatest element among the remaining elements in each iteration. Merge sort divides the task into subtasks on smaller inputs, starting with sorting the first and second halves of the given array, and then merging the sorted subarrays.

We are about to present another $\Theta(n^2)$-sorting algorithm, whose correctness is not so obvious. The algorithm was developed by Stanley P. Y. Fung, [Fun21], who coined its name - "ICan'tBelieveItCanSort" - due to the surprise of having such a simple sorting algorithm. It's worth mentioning that, despite its simplicity, Fung came up with this algorithm in 2021.

```
  Result: Sorting A_1, A_2, ..A_n
1 for i ∈ [n] do
2 |  for j ∈ [n] do
3 |  |  if A_i < A_j then
4 |  |  |  swap A_i ↔ A_j
5 |  |  end
6 |  end
7 end
```

Algorithm 11: "ICan'tBelieveItCanSort" alg.

Claim 5.1.1. *After the ith iteration, $A_1 \leq A_2 \leq A_3 .. \leq A_i$ and A_i is the maximum of the whole array.*

Proof. By induction on the iteration number i.

1. Base. For $i = 1$, it is clear that when j reaches the position of the maximum element, an exchange will occur and A_1 will be set to be the maximum element. Thus, the condition on line (3) will not be satisfied for the remaining j-iterations of the inner loop. Therefore, at the end of the first iteration, A_1 is indeed the maximum.

2. Assumption. Assume the correctness of the claim for any $i' < i$.

3. Step. Consider the ith iteration. And observe that if $A_i = A_{i-1}$ then A_i is also the maximal element in A, namely no exchange will be made in the ith iteration, yet $A_1 \leq A_2 \leq .. \leq A_{i-1}$ by the induction assumption, thus $A_1 \leq A_2 \leq .. \leq A_{i-1} \leq A_i$ and A_i is the maximal element, so the claim holds in the end of the iteration. If $A_i < A_{i-1}$ then there exists $k \in [1, i-1]$ such that $A_k > A_i$. Set k to be the minimal position for which the inequality holds. For convenience, denote by $A^{(j)}$ the array in the beginning of the jth iteration of the inner loop. And let's split into cases according to j value.

   (a) $j < k$ By definition of k, for any $j < k$, $A_j^{(1)} < A_i^{(1)}$, Hence in the first $k-1$ iterations no exchange will be made and we can conclude that $A_l^{(j)} = A_l^{(1)}$ for any $l \in [n]$ and $j \leq k$.

   (b) $j \geq k$ and $j \leq i$, We claim that for each such j an exchange will always occur. (The proof is given below.)

   Claim 5.1.2. *For any $j \in [k, i]$ we have that in the end of the jth iteration:*

   - $A_j^{(j+1)} = A_i^{(j)}$.
   - $A_i^{(j+1)} = A_j^{(j)} = A_j^{(1)}$.
   - *For any $l > j$ and $l \neq i$ we have $A_l^{(j+1)} = A_l^{(1)}$.*

   (c) $j > i$, so we know that $A_i^{(i+1)}$ is the maximal element in A. Therefore, for any j, it holds that $A_i^{(i+1)} \geq A_j^{(i)}$. It follows that no exchange would be made and $A_i^{(j)}$ will remain the maximum til the end of the inner loop. Thus for any $j > i$:

   $$A_i^{(j)} = A_i^{(j-1)} = .. = A_i^{(i+2)} = A_i^{(i+1)} = A_{i-1}^{(i)} = A_{i-1}^{(0)} = \max A$$

   And

   $$\begin{aligned} & A_1^{(j)}, A_2^{(j)}, ..A_{k-1}^{(j)}, A_k^{(j)}, A_{k+1}^{(j)}, ..A_{i-1}^{(j)}, A_i^{(j)}, A_{i+1}^{(j)}, A_{i+2}^{(j)}, A_{i+3}^{(j)}.. \\ = & A_1^{(0)}, A_2^{(0)}, ..A_{k-1}^{(0)}, A_i^{(0)}, A_k^{(0)}, ..A_{i-2}^{(0)}, A_{i-1}^{(0)}, A_{i+1}^{(0)}, A_{i+2}^{(0)}, A_{i+3}^{(0)}.. \end{aligned}$$

   In particular, for $j = n+1$ (Note that there is no $n+1$th iteration). Clearly, the inequalities are satisfied and we are done.

□

Proof of Claim 5.1.2. Observe that the third section holds trivially by the definition of the algorithm. It doesn't touch any position greater than j in the first j iterations (inner loop) except the ith position. So we have to prove only the first two bullets. Again, we are going to prove them by induction on j.

1. Base. $A_k^{(1)}$ is greater than A_i, and by the above case, we have that at the beginning of the kth iteration $A_k^{(k)} = A_k^{(1)}, A_i^{(k)} = A_k^{(1)}$. Therefore, the condition on line (3) is satisfied, an exchange is made, and $A_k^{(k+1)} = A_i^{(k)} = A_i^{(1)}$ and $A_i^{(k+1)} = A_k^{(k)}$.

2. Assumption. Assume the correctness of the claim for any $k \leq j' < j \leq i$.

3. Step. Consider the $j \in (k, i]$ iteration. By the induction assumption, we have that $A_{j-1}^{(j)} = A_i^{(j-1)}$ and $A_i^{(j)} = A_{j-1}^{(j-1)} = A_{j-1}^{(1)}$. On the other hand, by the induction assumption of Claim 5.1.1, $j-1 < i \Rightarrow A_{j-1}^{(1)} \leq A_j^{(1)}$. Combining the third bullet, we obtain that:

$$A_j^{(j)} = A_j^{(1)} \geq A_{j-1}^{(1)} = A_i^{(j)}$$

And therefore, either there is an inequality and an exchange is made or there is equality. In both cases, after the jth iteration, we have $A_j^{(j+1)} = A_i^{(j)}$ and $A_i^{(j+1)} = A_j^{(j)} = A_j^{(1)}$.

□

5.2 Master Theorem

Let $\alpha > 2, \beta > 0, \gamma > 0$ satisfying $2^\alpha < \beta$, define the following running time function:

$$T(n) = \beta T(n - \alpha) + n^\gamma$$

Bound T asymptotically tight.

Solution Let us define $S(m) = T(\log m)$. Thus, the recursive relation expands to:

$$\begin{aligned} S(m) = T(\log m) &= \beta T(\log m - \alpha) + \log^\gamma(m) \\ &= \beta T(\log m - \log 2^\alpha) + \log^\gamma(m) \\ &= \beta T(\log\left(\frac{m}{2^\alpha}\right)) + \log^\gamma(m) = \beta S(\frac{m}{2^\alpha}) + \log^\gamma(m) \end{aligned}$$

We observe that $2^\alpha > 1$ and therefore we can use the generalized master theorem. Given that $2^\alpha < \beta$, we have that $m^{\log_{2^\alpha}(\beta)}$ is a (generalized) polynomial with positive degree. Hence, there exists a positive ε such that $\log^\gamma(m) = O(m^{\log_{2^\alpha}(\beta)-\varepsilon})$. Therefore, by the master theorem, we conclude that:

$$S(m) = \Theta(m^{\log_{2^\alpha}(\beta)}) \Rightarrow T(n) = \Theta(2^{n \log_{2^\alpha}(\beta)})$$

Chapter 6

Linear Time Sorts and Lower Bounds in the Comparison Model.

6.1 Heapsort (David's group).

We will start by introducing the heap-sort algorithm and providing a proof of its correctness.

```
1 A ← Build-Heap(A)
2 for i ∈ [n] do
3 |   swap A_1 ↔ A_{n−i+1}
4 |   heapsize(A) ← n − i
5 |   heapify(A, 1)
6 end
7 return A
```

Algorithm 12: Heap-sort(A)

Correctness. We are going to prove the following statement.

Claim 6.1.1. *At the end of the ith iteration, $A_{n-i+1}, A_{n-i+2}, ..A_n$ are the i largest elements of A placed in order, and $A_1, A_2, ..A_{n-i}$ is a maximum heap.*

Proof. By induction.

1. Base. A_n is set in line (3) to be the root of the heap, and therefore is the maximum of A. After line (4), A_1 is the parent of the heap's roots as line (4) excludes A_n from the heap (So the heap inequality holds for any $j \in [2, (n-1)/2]$). Therefore, by the correctness of heapify, we get that after line (5), $A_1, A_2, ..A_{n-1}$ is a heap.

2. Assumption. Assume the correctness of the claim for any $i' < i$.

3. Step. Consider the ith iteration. By the induction assumption, A_1 is a root of the heap $A_1, A_2, ..A_{n-i+1}$ and therefore is their maximum. So after the swapping in line (3), we get that A_{n-i+1} is the element which is greater than

$n-i$ elements in A. By using the induction assumption again, we know that it is also less than $A_{n-i+2}, A_{n-i+3}, ..A_n$, so after line (3) and by the fact that $A_{n-i+2}, A_{n-i+3}, ..A_n$ are the $i-1$ largest elements placed in order, we have that $A_{n-i+1}, A_{n-i+2}, A_{n-i+3}, ..A_n$ are the i largest elements placed in order.

By repeating the same arguments in the base case, we can conclude, based on the correctness of heapify, that after line (5), A_1 is either the root of a heap or the heap inequality did not hold for some $i \in [2, (n-i)/2]$. In the latter case, this would contradict the induction assumption (since before line (3), $A_1..A_{n-i+1}$ were heaps).

□

6.2 Linear Time Sorts

Counting sort. Counting sort assumes that each of the n input elements is an integer with a size at most k. It runs in $\Theta(n+k)$ time, so when $k = O(n)$, counting sort runs in $\Theta(n)$ time. It first determines, for each input element x, the number of elements less than or equal to x. Then, it uses this information to place element x directly into its position in the output array. For example, if there are 17 elements less than or equal to x, then x will be placed in position 17. However, we need to make some adjustments to this method to handle cases where multiple elements have the **same value**. We do not want all of them to end up in the same position.

let B and C be new arrays at size n and k
for $i \in [0, k]$ **do**
    $C_i \leftarrow 0$
end
for $j \leftarrow [1, n]$ **do**
    $C_{Aj} \leftarrow C_{A_j} + 1$
end
for $i \in [1, k]$ **do**
    $C_i \leftarrow C_i + C_{i-1}$
end
for $j \in [n, 1]$ **do**
    $B_{C_{A_j}} \leftarrow A_j$
    $C_{A_j} \leftarrow C_{Aj}-1$ // to handle duplicate values
end
return B

Algorithm 13: Counting-sort(A, n, k)

Notice that the Counting sort can beat the lower bound of $\Omega(n \log n)$ only because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the elements to index into an array.

An important property of the counting sort is that it is **stable**.

Stable Sort.

We will say that a sorting algorithm is stable if elements with the same value appear in the output array in the same order as they do in the input array.

Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

Radix sort Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column, a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

Note 1: That introduction was copied word by word from a web source. Do not use for commercial purposes.

The Radix-sort procedure assumes that each element in the array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

```
for i ∈ [1, d] do
    use a stable sort to sort array A on digit i
end
```

Algorithm 14: radix-sort(A, n, d)

Correctness Proof. By induction on the column being sorted.

- Base. Where $d = 1$, the correctness follows immediately from the correctness of our base sort subroutine.
- Induction Assumption. Assume that Radix-sort is correct for any array of numbers containing at most $d - 1$ digits.
- Step. Let A' be the algorithm output. Consider $x, y \in A$. Assume without losing generality that $x > y$. Denote by x_d, y_d their d-digit and by $x_{/d}, y_{/d}$ the numbers obtained by taking only the first $d - 1$ digits of x, y. Separate in two cases:
  - If $x_d > y_d$ then a scenario in which x appear prior to y is imply contradiction to the correctness of our subroutine.
  - So consider the case in which $x_d = y_d$. In that case, it must hold that $x_{/d} > y_{/d}$. Then the appearance of x prior to y either contradicts the assumption that the base algorithm we have used is stable or that x appears before y at the end of the $d - 1$ iteration. Which contradicts the induction assumption.

The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit lies in the range 0 to k–1 (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes $\Theta(n + k)$ time. There are d passes, and so the total time for radix sort is $\Theta\left(d(n + k)\right)$.

Note 2: We will return to analyze the expected (average) running time after the lecture on probability.

Bucket sort. **[COMMENT] Only if there is time.** Bucket sort divides the interval [0, 1) into n equal-sized subintervals, or buckets, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over [0, 1), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

```
let B[0 : n − 1] be a new array
for i ← [0, n−1] do
│ make B_i an empty list
end
for i ← [1, n] do
│ insert A_i into list B_⌊nA_i⌋]
end
for i ← [0, n−1] do
│ sort list B_i
end
concatenate the lists B_0, B_1, .., B_{n−1} together and
return the concatenated lists
```

Algorithm 15: bucket-sort(A, n)

6.3 Sorting in Comparison Model.

We have learned in the lecture that the runs of any sorting algorithm, which does not assume anything about the input's structure except the ability to compare elements in pairs, can be modeled as a binary tree. At each node, the algorithm compares two elements and, based on the result, moves to either the left or right child. Eventually, we reach the leaf of the tree and output the sorted elements. Notice that the nodes only exist in our imagination; the algorithm is not aware of their existence. We will name that tree the comparison tree.

The height of the comparison tree is (at least) the running time of the algorithm. We are going to demonstrate how to use these ideas to lower-bound the running time of the find-peak problem in the comparison model.

Example 6.3.1. Prove that no algorithm can find a peak in less than $\Theta\left(\log n\right)$ time.

Proof. Consider the following inputs for the problem, A^j will be a tringle that get's is point at position j. Namely:

$$A_i^j = \begin{cases} i & i < j \\ j + j - i & \text{else} \end{cases}$$

Now, let's assume, in contradiction, that there exists an algorithm $\mathcal{A}$ in the comparison model that runs in less than $\log n$ time. This implies that the comparison tree representation of its running has a height less than $\log n$ and the number of its leaves, each associated with a possible output, is less than n. Remember that we have n inputs of the form A^j.

By applying the pigeonhole principle, we can conclude that there are distinct j and j' such that $\mathcal{A}$ reaches the same leaf when given A^j and $A^{j'}$ as inputs. Since their peaks are set at different positions, it follows that the algorithm will output a wrong answer for at least one of them. □

Example 6.3.2. Prove that no algorithm, in the comparison model, can merge two sorted arrays in time $\Theta(n^{1-\varepsilon})$ for some $\varepsilon > 0$.

Proof. Assuming, for contradiction, that there exists an algorithm $\mathcal{A}$ in the comparison model that merges two given sorted arrays in time $\Theta\left(n^{1-\varepsilon}\right)$. Now, consider the sorting algorithm obtained by replacing the merge subroutine in merge-sort with $\mathcal{A}$. On one hand, our new algorithm is a sorting algorithm in the comparison model (otherwise it would contradict the correctness of $\mathcal{A}$), while on the other hand, its running time is equal to:

$$T(n) = n^{1-\varepsilon} + 2T(n/2) \Rightarrow T(n) = \Theta(n)$$

This contradicts our $\Omega(n \log n)$ lower bound for sorting in the comparison model. □

6.4 Binaries Trees and How to Encode Them. [COMMENT] Only if there is time.

We have already seen, in heaps, that organizing our data in a graph-like structure can offer a speed advantage. For future applications, and in particular for maintaining data in sorted order, we will have to encode our data using binary trees. These trees may not be almost complete and also have to support pointer manipulations, specifically placing a binary tree as a left or right subtree of a given node. To enable this, we will have to treat the **right**, **left**, and **parent** as variables, in contrast to heaps where they are determined completely by the node index. We begin this section by stating definitions.

Definition 6.4.1.

1. Binary Tree: A tree in which any vertex has at most two children.
2. A descendant of vertex x is a vertex in the subtree whose root is x. A left descendant of vertex x is either a vertex in the subtree whose root is the left child of x, or x's left child.
3. An ancestor of x is a vertex to which x belongs as a descendant.
4. A leaf is a vertex without children.
5. Height of vertex x is the length of the longest simple path (without cycles) between x and one of the leaves.
6. Height of the tree is the height of its root, which is usually denoted by h.

We encode a binary tree by associating a field to each vertex x, representing its right, left children, and parent. We use the notation x.left to refer to the left child of x, although the physical implementation may differ conceptually. For example, the way binary trees are implemented in Cormen is through 4 arrays. The first stores the value of x, while the others store pointers of specific types. For instance, the array LEFT, where LEFT.x stores the left child of x.

If nothing else has been mentioned, then we can assume that we can add additional fields to the vertices.

6.5 Binary Search Trees.

Binary search tree (BST) is a binary tree which any node x of it:

1. Contains a field key, storing a number x.key.
2. Any left descendant y of x satisfies $y.\text{key} \leq x.\text{key}$.
3. Any right descendant y of x satisfies $y.\text{key} \geq x.\text{key}$.

Question. Let T be a binary search tree, Where are the minimum and maximum values of T? (most left and right nodes).

Definition 6.5.1. Let T be a binary search tree, and let x be a node belonging to it. The predecessor of x will be defined as a vertex y such that y.key $\leq x$.key and y.key is maximal among the nodes satisfying this condition. If we were to set the values of T in sorted order, then the predecessor of x would be located on its left. The successor of x will be defined as y, where x is the predecessor of y.

Functionalitiy of BST.

1. Search(T, key): returns a pointer to the vertex whose key equals key.
2. Min(T): returns a pointer to the vertex with the minimum value in T.
3. Max(T): returns a pointer to the vertex with the maximum value in T.
4. Predecessor(x): returns a pointer to the predecessor of x.
5. Successor(x): returns a pointer to the successor of x.
6. Insert(T, key): inserts key into T (creates a new vertex).
7. Delete(T, x): removes x from T.
8. Inorder(T): outputs T's keys in sorted order.

Example 6.5.1. Questions from past exams.

1. Write an algorithm that, given a binary search tree T, returns a max heap containing all the elements of T. What is the lower bound for this problem? Explain.
2. Write an algorithm that, given a heap H, returns a binary search tree containing all the elements of H. What is the lower bound for this problem? Explain.

Solution.

1. Remember that any array sorted in reverse order is also a max heap. This is because if $A_i \geq A_j$ for any $i < j$, then it follows that $A_i \geq A_{2i}, A_{2i+1}$ for any i. Therefore, we will output the values of T in reverse order. We do this by applying Inorder in reverse, as given in Algorithm 16. The running time is $\Theta(n)$ and of course it is also the lower bound (as we must read all the inputs for printing it).
2. We will initialize an empty binary search tree and then add the elements to it one by one. Each insertion will cost the current height of the tree, so it might sum up to $\Theta(n^2)$. However, next week we will see how those insertions could be done in a way that preserves the balance of the tree, namely guaranteeing that the height of the tree will remain logarithmic.

Example 6.5.2. Adding fields to BSTs. We would like to support k-smallest element extraction. Suggest a field that will be used for computing the k-smallest element, explain how to maintain it, and write an algorithm that extracts the k-smallest element.

Data: r - a vertex in BST
if *r.right is not empty/Null* **then**
    Reverse-Order(r.right)
end
Print r.key
if *r.left is not empty/Null* **then**
    Reverse-Order(r.left)
end

Algorithm 16: Reverse-Order

Solution. We will associate with each vertex of the tree a field that counts the number of nodes whose keys are lower than it (the size of its left subtree). Let's denote it as left-size. The code is given in Algorithm 17. Guideline for proving correctness: Consider the root, and let's separate into cases based on how many elements the root is greater than.

1. If the root is strictly greater than $k-1$ elements, then the k-smallest element is in its left subtree and is also the $k-1$ smallest element in that subtree.

2. If the root is greater than strictly fewer than $k-1$ elements, then it is clear that the k-smallest element is also greater than it and located in its right subtree. However, in contrast to the previous case, here the order of the k-smallest element of the whole tree in the subtree will be the substitution between k and the number of nodes in the $\{$ left-subtree $\cup$ root $\}$.

3. The third case is when the root is greater than exactly $k-1$ elements, which by definition sets it as the k-smallest element.

In the first two cases, we are calling Get-Order on trees (BST with our additional field) with smaller height. Therefore, it is clear how one would prove correctness by induction. We will see in the next recitation how to maintain the tree, update the field, and guarantee that the height of the tree will remain logarithmic.

Data: r - a vertex in BST, k-stat parameter
$m \leftarrow r$.left-size
if $m = k-1$ **then**
    return r.key
end
if $m > k-1$ **then**
    return Get-Order(r.left, k)
end
if $m < k-1$ **then**
    return Get-Order(r.right, $k-(m+1)$)
end

Algorithm 17: Get-Order

6.6 Appendix - BST Methods Source: A Mixture of Cormen, Wikipedia, and Codeforces

Data: T - tree, key - key to search for
Result: pointer to vertex with key equal to key

```
if T is empty then
    return null;
end
if T.key equals key then
    return T;
end
if key is less than T.key then
    return Search(T.left, key);
end
else
    return Search(T.right, key);
end
```

Algorithm 18: Search

Data: T - tree
Result: pointer to vertex with minimum value in T

```
if T is empty then
    return null;
end
if T.left is empty then
    return T;
end
return Min(T.left);
```

Algorithm 19: Min

Data: T - tree
Result: pointer to vertex with maximum value in T
if T *is empty* **then**
  return null;
end
if T*.right is empty* **then**
  return T;
end
return Max(T.right);

Algorithm 20: Max

Data: x - vertex
Result: pointer to predecessor of x
if x*.left is not empty* **then**
  return Max(x.left);
end
$y \leftarrow x$.parent;
while y *is not empty and* x *is* y*.left* **do**
  $x \leftarrow y$;
  $y \leftarrow y$.parent;
end
return y;

Algorithm 21: Predecessor

Data: x - vertex
Result: pointer to successor of x
if x*.right is not empty* **then**
  return Min(x.right);
end
$y \leftarrow x$.parent;
while y *is not empty and* x *is* y*.right* **do**
  $x \leftarrow y$;
  $y \leftarrow y$.parent;
end
return y;

Algorithm 22: Successor

```
Data: T - tree, key - key to insert
Result: inserts key into T
newNode ← create new vertex with key key;
y ← null;
x ← T;
while x is not empty do
    y ← x;
    if key is less than x.key then
        x ← x.left;
    end
    else
        x ← x.right;
    end
end
newNode.parent ← y;
if y is null then
    T ← newNode;
end
else if key is less than y.key then
    y.left ← newNode;
end
else
    y.right ← newNode;
end
```

Algorithm 23: Insert

```
Data: T: The input tree
if T is not empty then
    Inorder(T.left);
    Output T.key;
    Inorder(T.right);
end
```

Algorithm 24: Inorder(T)

Chapter 5

DAST. Exercise 4.

5.1 Stability.

For each of the following sorts, prove or disprove whether they are stable.

1. Heap sort.
2. Bubble sort.
3. Counting sort.

5.2 Sorts That Assume Input Properties.

In any of the following sections, we make another assumption about the input. For each section, write an algorithm that uses the assumption to sort the inputs and compute its running time. You don't have to provide a correctness proof.

1. $x_1, x_2, ..., x_n \in \mathbb{R}$ such that $\max_i\{x_i\} - \min_i\{x_i\} \leq 5n$ and $|x_i - x_j| > 1$ for any $i \neq j$.
2. $x_1, x_2, ..., x_n \in \mathbb{R}$ such that $\max_i\{x_i\} - \min_i\{x_i\} \leq 5n$ and $|x_i - x_j| > \Delta$ for any $i \neq j$, but it is unknown what Δ is. The running time should depend on Δ.

5.3 Camels.

Bob is willing to send Alice camels, $c_1, c_2, ..., c_n$ in a specific order. Unfortunately for him, Eve doesn't like him. Hence, she decides to spoil their order by stacking camels as follows: Eve chooses camel c_i and a position j, and then moves the ith camel c_i between c_j and c_{j+1}, She can move as many camels as she wishes. Alice, known for her strictness, wouldn't accept camels standing out of order. Luckily, Bob is allowed to send the camels again in their original order for five times, and Eve cannot choose the same camel twice. In other words, if Eve chooses to move c_i in the first attempt, then she cannot move it in any of the other four attempts.

Alice records the positions of the camels every time they come. Write an algorithm that will help Alice restore the order. (Prove correctness and compute the running time.)

5.4 Predecessor in Binary Search Tree

1. Write pseudocode for the predecessor subroutine.
2. Prove its correctness.

Chapter 7

Probability.

7.1 Probability Spaces.

Definition 7.1.1. A probability space is defined by a tuple (Ω, P), where:

1. Ω is a set, called the sample space. Any element $\omega \in \Omega$ is an atomic event. Conceptually, we think of atomic events as possible outcomes of our experiment. Any subset $A \subset \Omega$ is an event.

2. P, called the probability function, is a function that assigns a number in $[0, 1]$ to any event, denoted as $P : 2^{\Omega} \to [0, 1]$, and satisfies:

   (a) For any event $A \subset \Omega$, $P(A) = \sum_{\omega \in A} P(\omega)$.

   (b) Normalization over the atomic events to 1, which means $\sum_{\omega \in \Omega} P(\omega) = 1$.

Example 7.1.1. Consider a dice rolling, where each of the faces is indexed by $1, 2, 3, 4, 5, 6$ and has an equal chance of being rolled. Therefore, our atomic events are associated with the rolling result, and P is defined as $P(\omega) = \frac{1}{6}$ for any such atomic event. An example of an event can be $A =$ "the dice falls on an even number". The probability of this outcome is:

$$P(A) = \sum_{\omega \in A} P(\omega) = P(\{2\}) + P(\{4\}) + P(\{6\}) = 3 \cdot \frac{1}{6} = \frac{1}{2}$$

Claim 7.1.1. *The probability function satisfies the following properties:*

1. $P(\emptyset) = 0$.

2. *Monotonicity: If* $A \subset B \subset \Omega$, *then* $P(A) \leq P(B)$.

3. *Union Bound:* $P(A \cup B) \leq P(A) + P(B)$.

4. *Additivity for disjoint events: If* $A \cap B = \emptyset$, *then* $P(A \cup B) = P(A) + P(B)$.

5. *Complementarity: Denote by* $\bar{A}$ *the complementary event of* A, *which means* $A \cup \bar{A} = \Omega$. *Then,* $P(\bar{A}) = 1 - P(A)$.

Example 7.1.2. Let's proof the additivity of disjointness property. Let A, B disjointness events, so $A \cap B = \emptyset$ then

$$\begin{aligned}P(A \cup B) &= \sum_{w \in A \cup B} P(w) \\ &= \overbrace{\sum_{w \in A, w \notin B} P(w)}^{P(A)} + \overbrace{\sum_{w \in B, w \notin A} P(w)}^{P(B)} + \overbrace{\sum_{w \in A, w \in B} P(w)}^{0} \\ &= P(A) + P(B)\end{aligned}$$

Definition 7.1.2. Let (Ω, P) be a probability space. A random variable X on (Ω, P) is a function $X : \Omega \to \mathbb{R}$. An indicator, is a random variable defined by an event $A \subset \Omega$ as follows

$$X(\omega) = \begin{cases} 1 & \omega \in A \\ 0 & \omega \notin A \end{cases}$$

Sometimes, we will use the notation $\{X = x\}$ to denote the event A such:

$$A = \{\omega : X(\omega) = x\} := \{X = x\}$$

Example 7.1.3. Consider rolling a pair of dice. Denote by $X : [6] \times [6] \to [6]$ the random variable that is set to be the result of the first roll. Let Y be defined in almost the same way, but setting the result of the second die. Namely, if we denote by $\{(i, j)\}$ the atomic event associated with sample i on the first die and j on the second die, then:

$$\begin{aligned}X(\{i, j\}) = i \\ Y(\{i, j\}) = j\end{aligned}$$

In addition, one can define the random variable z as the sum, $Z = X + Y$. Since the sum is also a function from Ω to $\mathbb{R}$, Z is also a random variable. An example of an indicator could be W, which gets 1 if $Z \in \{2, 7, 8\}$.

Example 7.1.4. Let X be an indicator of event A. Then $1 - X$ is the indicator of $\bar{A}$.

$$1 - X(\omega) = \begin{cases} 0 & \omega \in A \Leftrightarrow \omega \notin \bar{A} \\ 1 & \omega \notin A \Leftrightarrow \omega \in \bar{A} \end{cases}$$

Definition 7.1.3. We will say that two events A, B are independent if:

$$P(A \cap B) = P(A) \cdot P(B)$$

Similarly we will say that random variables $X, Y : \Omega \to \mathbb{R}$ are independent if for any $x \in$ Im X and $y \in$ Im Y:

$$P(X = x \cap Y = y) = P(X = x) \cdot P(Y = y)$$

Example 7.1.5. X, Y defined in Example 7.1.3 are independent.

$$\begin{aligned}P(\{X = i\} \cap \{Y = j\}) &= \sum_{i' = i \text{ and } j' = j} P(\{(i', j')\}) = P(\{(i, j)\}) \\ &= \frac{1}{36} = \frac{1}{6} \cdot \frac{1}{6} = P(X = i) P(Y = j)\end{aligned}$$

Example 7.1.6. Let A and B be independent events. Then, $\bar{A}$ and B are also independent events, since:

$$\begin{aligned} P(B) &= P(B \cap \Omega) = P(B \cap (A \cup \bar{A})) = P((B \cap A) \cup (B \cap \bar{A})) \\ &= P(B \cap A) + P(B \cap \bar{A}) = P(B)P(A) + P(B \cap \bar{A}) \\ \Rightarrow P(B \cap \bar{A}) &= P(B)(1 - P(A)) = P(B)P(\bar{A}) \end{aligned}$$

Example 7.1.7. Let X and Y be indicators of independent events A and B. Then $P(X \cdot Y = 1) = P(X = 1) \cdot P(Y = 1)$. The proof is left as an exercise.

7.2 Throwing Keys to Cells.

Example 7.2.1. Imagine that following experiment, we have m cells and n keys (balls, numbers, or your favorite object type). We throw each of the keys independently into the cells. The cells are identical, so the probability of hitting any of them is the same, $1/m$. We would like to analyze how the capacity of the cells is distributed.

1. What is the probability that the first and the second keys will be thrown to the first cell? What is the probability that the first and the second keys will be thrown to the same cell?

2. What is the probability that in the first cell there is exactly one key?

Let us define the indicator X_i^j which indicate that the jth key fallen into the ith cell.

1. So first we been asked whether $X_1^1 \cdot X_1^2 = 1$, Since this happens only if both $X_1^1 = 1$, $X_1^2 = 1$ then by independently we have that:

$$\begin{aligned} P(X_1^1 \cdot X_1^2 = 1) &= P(X_1^1 = 1 \cap X_1^2 = 1) \\ &= P(X_1^1 = 1) \cdot P(X_1^2 = 1) = \frac{1}{m^2} \end{aligned}$$

   Now, to answer if the first and second keys fall into the same cell, we need to check if there exists an i such that $X_i^1 \cdot X_i^2 = 1$. Observes that for any different i and i', the X_i^j and $X_{i'}^j$ are indicators of disjoint events. This is because j cannot be in both the i and i' cells. Therefore, $X_i^1 \cdot X_i^2$ and $X_{i'}^1 \cdot X_{i'}^2$ are also indicators of disjoint events. Thus:

$$\begin{aligned} P(\exists i : X_i^1 \cdot X_i^2 = 1) &= P(\bigcup_i X_i^1 \cdot X_i^2 = 1) \\ &= \sum_i P(X_i^1 \cdot X_i^2 = 1) = m \cdot \frac{1}{m^2} = \frac{1}{m} \end{aligned}$$

   We are basically done. However, we want to present the same calculation in a different notation that will be useful for computing expectations later on. Note that the random variable that counts "how many" cells both the first and the second fall into is $\sum_i X_i^1 \cdot X_i^2$. In other words, the sum can be either 0 if the keys fall into different cells, or 1 if they both fall into the same cell.

2. The event that only the jth key falls into the first cell matches to

$$\left\{X_1^j \prod_{j \neq j'} \left(1 - X_1^{j'}\right) = 1\right\}$$

Therefore, due to the disjointness of $1 - X_1^{j'}$ and $X_1^{j'}$, the indicator for the first cell containing exactly one key is:

$$\left\{\sum_j X_1^j \prod_{j \neq j'} \left(1 - X_1^{j'}\right) = 1\right\}$$

Since the terms in the sum are disjoint and the products are products of independent indicators, we have:

$$P\left(\sum_j X_1^j \prod_{j \neq j'} \left(1 - X_1^{j'}\right) = 1\right) = \sum_j P\left(X_1^j \prod_{j \neq j'} \left(1 - X_1^{j'}\right) = 1\right)$$
$$= m \cdot \frac{1}{m}\left(1 - \frac{1}{m}\right)^{n-1} = \left(1 - \frac{1}{m}\right)^{n-1}$$

Definition 7.2.1. Let $X : \Omega \to \mathbb{R}$ be a random variable, the expectation of X is

$$\mathbf{E}[X] = \sum_{\omega \in \Omega} X(\omega)P(\omega) = \sum_{x \in \text{ Im } X} xP(X = x)$$

Observes that if P is distributed uniformly, then the expectation of X is just the arithmetic mean:

$$\mathbf{E}[X] = \sum_{\omega \in \Omega} X(\omega)P(\omega) = \frac{1}{|\Omega|}\sum_{\omega \in \Omega} X(\omega)$$

Claim 7.2.1. *The expectation satisfies the following properties:*

1. *Monotonic, If $X \leq Y$ (for any $\omega \in \Omega$) then $\mathbf{E}[X] \leq \mathbf{E}[Y]$.*
2. *Linearity, for $a, b \in \mathbb{R}$ it holds that $\mathbf{E}[aX + by] = a\mathbf{E}[X] + b\mathbf{E}[Y]$.*
3. *Independently, if X, Y are independent, then $\mathbf{E}[X \cdot Y] = \mathbf{E}[X] \cdot \mathbf{E}[Y]$.*
4. *For any constant $a \in \mathbb{R}$ we have that $\mathbf{E}[a] = a$.*

Proof. 1. Monotonic, if $X \leq Y$ then :

$$\mathbf{E}[X] = \sum_{\omega \in \Omega} X(\omega)P(\omega) \leq \sum_{\omega \in \Omega} Y(\omega)P(\omega) = \mathbf{E}[Y]$$

2. Linearity,

$$\begin{aligned}\mathbf{E}[aX + bY] &= \sum_{\omega \in \Omega} \left(aX(\omega) + bY(\omega)\right)P(\omega) \\ &= a\sum_{\omega \in \Omega} X(\omega)P(\omega) + b\sum_{\omega \in \Omega} Y(\omega)P(\omega)\end{aligned}$$

3. Independently,

$$\begin{aligned}
\mathbf{E}[XY] &= \sum_{x,y\in \text{ Im } X\times \text{ Im } Y} xyP(X=x\cap Y=y)\\
&= \sum_{x,y\in \text{ Im } X\times \text{ Im } Y} xyP(X=x)P(Y=y)\\
&= \sum_{x\in \text{ Im } X}\sum_{y\in \text{ Im } Y} xyP(X=x)P(Y=y)\\
&= \sum_{x\in \text{ Im } X} xP(X=x)\sum_{y\in \text{ Im } Y} yP(Y=y)\\
&= \sum_{x\in \text{ Im } X} xP(X=x)\mathbf{E}[Y]\\
&= \mathbf{E}[X]\,\mathbf{E}[Y]
\end{aligned}$$

4. Let X be the random variable which is also the constant function $X(\omega)=a$ for any $\omega\in\Omega$. Then we have that

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{\omega\in\Omega} X(\omega)P(\omega)\\
&= \sum_{\omega\in\Omega} aP(\omega) = a\cdot 1 = a
\end{aligned}$$

□

Example 7.2.2. Let X be an indicator of event A, what are $\mathbf{E}[X]$ and $\mathbf{E}\left[X^2\right]$? Recall that $X(\omega)=1$ only if $\omega\in A$ and 0 otherwise, thus:

$$X^k(\omega)=\begin{cases}1^k=1 & \omega\in A\\ 0^k=0 & \text{else}\end{cases} \Rightarrow X^k(\omega)=X(\omega)$$

Therefore,

$$\mathbf{E}\left[X^k\right]=\sum_{\omega\in\Omega} X^k(\omega)P(\omega)=\sum_{\omega\in\Omega} X(\omega)P(\omega)=\mathbf{E}[X]$$

Example 7.2.3. Consider the experiment of throwing keys into cells again. What is the expected number of keys that fell into the same cell as the first key? The indicator of the event j and 1 falling into the same cell is given by $\sum_i X_i^1 X_i^j$ and it remains to sum over all the j's. So:

$$\mathbf{E}\left[\sum_i\sum_j X_i^1X_i^j\right]=\sum_i\sum_j\mathbf{E}\left[X_i^1X_i^j\right]$$

$$\begin{aligned}
&=\sum_i\sum_{j\neq 1}\mathbf{E}\left[X_i^1\right]\mathbf{E}\left[X_i^j\right]+\overbrace{\sum_i\mathbf{E}\left[X_i^1\right]}^{j=1}\\
&=m\cdot(n-1)\frac{1}{m^2}+m\cdot\frac{1}{m}=\frac{n-1}{m}+1
\end{aligned}$$

Note 1: Despite the ease of computing the expectation, calculating the exact probability of $\sum_i\sum_j X_i^1X_i^j=L$ for some arbitrary L is a difficult task.

7.3 Running Time as a Random Variable.

Randomness might appears in algoritmic context in two main cases, in the first the algorithm might behave randomly, means that it flips coins and decided what to do conditioning on the outcomes. In the second case, the input might distributed according to probability function. In both cases the result and running time of the algorithm are random variable. And it's interesting to ask what is the expected running time.

Let us introduce an example for the first case. We are given an array $A_1, A_2, \ldots, A_n$ and are asked to find the k smallest elements in it. Here, we are going to suggest a random algorithm that is expected to return in linear time, even if we do not make any assumptions about the input, particularly how it is distributed.

Result: returns the k smallest element in $A_1 \ldots A_n \in \mathbb{R}^n$

1 **if** *left* $=$ *right* **then**
2   **return** A_{left}
3 **end**
4 pivot $\leftarrow$ select random pivot in [left, right]
5 pivot $\leftarrow$ partition(A, left, right, pivot)
6 **if** k = *pivot* **then**
7   **return** A_k
8 **end**
9 **else if** $k <$ *pivot* **then**
10   right $\leftarrow$ pivot - 1
11 **end**
12 **else**
13   left $\leftarrow$ pivot + 1
14     $k \leftarrow k -$ pivot
15 **end**
16 **return** call select(A, left, right, k)

Algorithm 25: select(A, left, right, k)

Consider the first call to 'select' and let X_m be the indicator for selecting the index of the mth smallest number on line (4). Notice that X_m and the running time of the recursive calls are independent random variables. Additionally, we will assume in induction that $T(n', k) \leq 2cn'$ for any $n' < n$. Therefore, the expected running time is:

$$\begin{aligned}
T(n,k) &= c \cdot n + \sum_{m<k} X_m \cdot T(n-m, k-m) \\
&\quad + X_k \cdot 1 + \sum_{m>k} X_m \cdot T(m-1, k) \\
\Rightarrow \mathbf{E}\left[T(n,k)\right] &\leq cn + 2c + \sum_{m<k} \mathbf{E}\left[X_m \cdot T(n-m, k-m)\right] \\
&\quad + \sum_{m>k} \mathbf{E}\left[X_m \cdot T(m-1, k)\right] \\
&\leq c \cdot n + 2c + 2c \sum_{m<k} \frac{n-m}{n} + 2c \sum_{m>k} \frac{m-1}{n} \\
&\leq c \cdot n + 2c + 2c \sum_{m<k} \frac{n-m}{n} + 2c \sum_{m>k} \frac{m}{n}
\end{aligned}$$

Claim 7.3.1. *The sum above is maximal when* $k = \lfloor n/2 \rfloor$.

Proof. We will prove that for $k = i+1$, the sum is greater than $k = i$ if $i < \lfloor n/2 \rfloor$. Denote $S_i = \sum_{m<i} \frac{n-m}{n} + \sum_{m>i} \frac{m}{n}$. Then, the substitution of $S_{i+1} - S_i$ becomes:

$$S_{i+1} - S_i = \frac{n-i-1}{n} - \frac{i}{n} = \frac{n-2i-1}{n}$$

And that quantity is positive for any $i < \lfloor n/2 \rfloor$. By symmetry, we obtain that for any $i > \lceil n/2 \rceil + 1$, the quantity $S_i - S_{i+1}$ is positive. Hence, $\lfloor n/2 \rfloor$ is a global maximum. ☐

Therefore, the expectation is bounded by:

$$\begin{aligned}
&\leq c \cdot n + 2c + 2c \sum_{m<\lfloor n/2 \rfloor} \frac{n-m}{n} + 2c \sum_{m>\lfloor n/2 \rfloor} \frac{m}{n} \\
&= c \cdot n + 2c + 2 \cdot 2c \sum_{m>\lfloor n/2 \rfloor} \frac{m}{n} \\
&= c \cdot n + 2c + 2 \cdot 2c \cdot \frac{(n/2) \cdot (n/2 - 1)}{2n} \\
&\leq cn + 2c + n/2 \cdot 2c \leq 2c \cdot n
\end{aligned}$$

Chapter 8

Heaps - Recitation 4

Apart from quantifying the resource requirement of our algorithms, we are also interested in proving that they indeed work. In this Recitation, we will demonstrate how to prove correctness via the notation of loop invariant. In addition, we will present the first (non-trivial) data structure in the course and prove that it allows us to compute the maximum efficiently.

Correctness And Loop Invariant.

In this course, any algorithm is defined relative to a task/problem/function, And it will be said correctly if, for any input, it computes desirable output. For example, suppose that our task is to extract the maximum element from a given array. So the input space is all the arrays of numbers, and proving that a given algorithm is correct requires proving that the algorithm's output is the maximal number for an arbitrary array. Formally:

Correctness.

We will say that an algorithm $\mathcal{A}$ (an ordered set of operations) computes $f : D_1 \rightarrow D_2$ if for every $x \in D_1 \Rightarrow f(x) = \mathcal{A}(x)$. Sometimes when it's obvious what is the goal function f, we will abbreviate and say that $\mathcal{A}$ is correct.

Other functions f might be including any computation task: file saving, summing numbers, posting a message in the forum, etc. Let's dive back into the maximum extraction problem and see how one should prove correctness in practice.

Task: Maximum Finding. *Given $n \in \mathbb{N}$ numbers $a_1, a_2, \cdots a_n \in \mathbb{R}$ write an Algorithm that returns their maximum.*

Consider the following suggestion. How would you prove it correct?

input: Array $a_1, a_2, ..a_n$
let $b \leftarrow a_1$
for $i \in [2, n]$ **do**
  $b \leftarrow \max(b, a_i)$
end
return b

Usually, it will be convenient to divide the algorithms into subsections and then characterize and prove their correctness separately. One primary technique is using the notation of Loop Invariant. Loop Invariant is a property that is characteristic of a loop segment code and satisfies the following conditions:

Loop Invariant.

1. Initialization. The property holds (even) before the first iteration of the loop.
2. Conservation. As long as one performs the loop iterations, the property still holds.
3. Termination. Exiting from the loop carrying information.

Let's denote by $b^{(i)}$ the value of b at line number 2 at the ith iteration for $i \geq 2$ and define $b^{(1)}$ to be its value in its initialization. What is the Loop Invariant here?
Claim. *"at the i-th iteration, $b^{(i)} = \max\{a_1 ... a_i\}$".*

Proof. Initialization, clearly, $b^{(1)} = a_1 = \max\{a_1\}$. Conservation, by induction, we have the base case from the initialization part, assume the correctness of the claim for any $i' < i$, and consider the ith iteration (of course, assume that $i < n$). Then:

$$b^{(i)} = \max\{b^{(i-1)}, a_i\} = \max\{\max\{a_1, ..a_{i-2}, a_{i-1}\}, a_i\} = \max\{a_1, ..a_i\}$$

And that completes the Conservation part. Termination, followed by the conservation, at the n iteration, $b^{(i)}$ is seted to $\max\{a_1, a_2..a_n\}$.

Task: Element finding. *Given $n \in \mathbb{N}$ numbers $a_1, a_2, \cdots a_n \in \mathbb{R}$ and additional number $x \in \mathbb{R}$ write an Algorithm that returns i s.t $a_i = x$ if there exists such i and* False *otherwise.*

```
input: Array a_1, a_2, ..a_n
1 for i ∈ [n] do
2     if a_i = x then
3         return i, a_i
4     end
5 end
6 return False
```

Correctness Proof. First, let's prove the following loop invariant.

Claim *Suppose that the running of the algorithm reached the i-th iteration, then $x \notin \{a_1..a_{i-1}\}$.* **Proof.** Initialization, for $i = 1$ the claim is trivial. Let's use that as the induction base case for proving Conservation. Assume the correctness of the claim for any $i' < i$. And consider the ith iteration. By the induction assumption, we have that $x \notin \{a_1..a_{i-2}\}$, and by the fact that we reached the ith iteration, we have that in the $i - 1$ iteration, at the line (2) the conditional weren't satisfied (otherwise, the function would return at line (3) namely $x \neq a_{i-1}$. Hence, it follows that $x \notin \{a_1, a_2..a_{i-2}, a_{i-1}\}$.

Separate to cases. First, consider the case that given the input $a_1..a_n$, the algorithm return False. In this case, we have by the termination property that $x \notin \{a_1..a_n\}$. Now, Suppose that the algorithm returns the pair (i, x), then it means that the conditional at the line (2) was satisfied at the ith iteration. So, indeed $a_i = x$, and the algorithm returns the expected output.

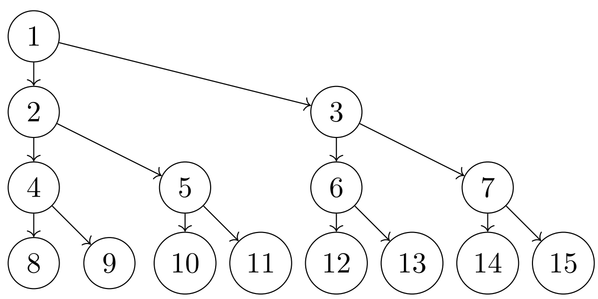
Leading Problem. find k largest elements. **[COMMENT]** Next year present the finding k-largest elements in $\Theta(n + k \log k)$ time as the goal of the recitation.

Heap

Let $n \in \mathbb{N}$ and consider the sequence $H = H_1, H_2 \cdots H_n \in \mathbb{R}\,(*)$. we will say that H is a Heap if for every $i \in [n]$ we have that: $H_i \leq H_{2i}, H_{2i+1}$ when we think of the value at indices greater than n as $H_{i>n} = -\infty$.

$$\Leftrightarrow$$

That definition is equivalent to the following recursive definition: Consider an almost complete binary tree where each node is associated with a number. Then, we will say that this binary tree is a heap if the root's value is lower than its children's values, and each subtree defined by its children is also a heap. There is a one-to-one mapping between these definitions by setting the array elements on the tree in order.



Checking vital signs. Are the following sequences are heaps?

1. 1,2,3,4,5,6,7,8,9,10 (Y)
2. 1,1,1,1,1,1,1,1,1,1 (Y)
3. 1,4,3,2,7,8,9,10 (N)
4. 1,4,2,5,6,3 (Y)

How much is the cost (running time) to compute the min of H? (without changing the heap). ($O(1)$). Assume that option 4 is our Superpharm Line. Let's try to imagine how we should maintain the line. After serving the customer at the top, what can be said on $\{H_2, H_3\}$? or $\{H_{i>3}\}$? (the second highest value is in $\{H_2, H_3\}$.)

Subtask: Extracting Heap's Minimum. *Let H be an Heap at size n, Write algorithm which return H_1, erase it and returns H', an Heap which contain all the remain elements.* **Solution:**

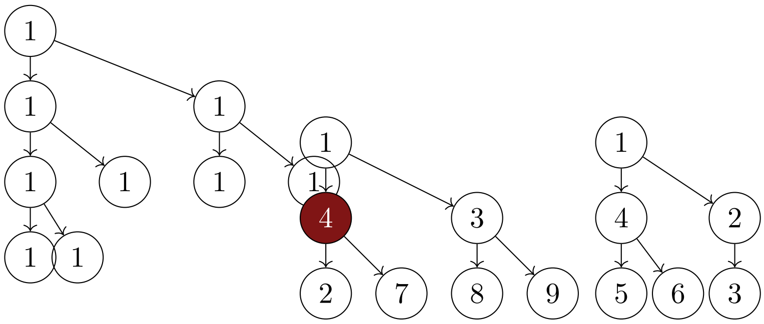


Figure 8.1: The trees representations of the heaps above. The node which fails to satisfy the Heap inequality is colorized.

input: Heap $H_1, H_2, ..H_n$

ret $\leftarrow H_1$

$H_1 \leftarrow H_n$

$n \leftarrow n - 1$

Heapify-down(1)

return ret

input: Array $a_1, a_2, ..a_n$

next $\leftarrow i$

left $\leftarrow 2i$

right $\rightarrow 2i + 1$

if *left* $< n$ *and* $H_{left} < H_{next}$ **then**

  next $\leftarrow$ left

end

if *right* $< n$ *and* $H_{right} < H_{next}$ **then**

  next $\leftarrow$ right

end

if $i \neq$ *next* **then**

  $H_i \leftrightarrow H_{\text{next}}$

  Heapify-down(next)

end

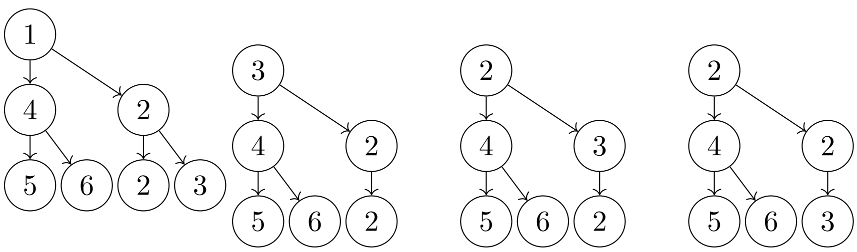


Figure 8.2: Running Example, Extract.

Claim. Assume that H satisfies the Heap inequality for all the elements except the root. Namely for any $i \neq 1$ we have that $H_i \leq H_{2i}, H_{2i+1}$. Then applying Heapify-down on H at index 1 returns a heap.

Proof. By induction on the heap size.

- Base, Consider a heap at the size at most three and prove for each by considering each case separately. (lefts as exercise).
- Assumption, assume the correctness of the Claim for any tree that satisfies the heap inequality except the root, at size $n' < n$.
- Induction step. Consider a tree at size n which and assume w.l.g (why could we?) that the right child of the root is the minimum between the triple. Then, by the definition of the algorithm, at line 9, the root exchanges its value with its right child. Given that before the swapping, all the elements of the heap, except the root, had satisfied the heap inequality, we have that after the exchange, all the right subtree's elements, except the root of that subtree (the original root's right child) still satisfy the inequality. As the size of the right subtree is at most $n-1$, we could use the assumption and have that after line (10), the right subtree is a heap.

  Now, as the left subtree remains the same (the values of the nodes of the left side didn't change), we have that this subtree is also a heap. So it's left to show that the new tree's root is smaller than its children's. Suppose that is not the case, then it's clear that the root of the right subtree (heap) is smaller than the new root. Combining the fact that its origin must be the right subtree, we have a contradiction to the fact that the original right subtree was a heap (as its root wasn't the minimum element in that subtle).

Question. How to construct a heap? And how much time will it take?

```
input: Array H = H_1..H_n
i ← ⌊(1/2)n⌋
while i > 1 do
    Heapify-down (H, i)
    i ← i − 1
end
return H_1..H_n
```

We can compute a simple upper bound on the running time of Build as follows. Each call to Heapify costs $O(\log n)$ time, and Build makes $O(n)$ such calls. Thus, the running time is $O(n \log n)$. This upper bound, though correct, is not as tight as it can be.

Let's compute the tight bound. Denote by U_h the subset of vertices in a heap at height h_i. Also, let $c > 0$ be the constant quantify the work that is done in each recursive step, then we can express the total cost as being bonded from above by:

$$T(n) \leq \sum_{h_i=1}^{\log n} c \cdot (\log n - h_i) \left| U_{h_i} \right|$$

By counting arguments, we have the inequality $2^{\log n - h_i} |U_i| \leq n$ (Proving this argument is left as an exercise). We thus obtain:

$$
\begin{aligned}
T(n) \leq & \sum_{h_i=1}^{\log n} c \cdot (\log n - h_i) \frac{n}{2^{logn-h_i}} = cn \sum_{j=1}^{\log n} 2^{-j} \cdot j \\
\leq & cn \sum_{j=1}^{\infty} 2^{-j} \cdot j
\end{aligned}
$$

And by the Ratio test for inifinte serires $\lim_{j\to\infty} \frac{(j+1)2^{-j-1}}{j2^{-j}} \to \frac{1}{2}$ we have that the serire covergence to constant. Hence: $T(n) = \Theta(n)$.

Heap Sort. Combining the above, we obtain a new sorting algorithm. Given an array, we could first Heapify it (build a heap from it) and then extract the elements by their order. As we saw, the construction requires linear time, and then each extraction costs $\log n$ time. So, the overall cost is $O(n \log n)$. Correction follows immediately from Build and Extract correction.

input: Array $H_1, H_2, ..H_n$
```
H ← build (x_1, x_2..x_n)
ret ← Array {}
for i ∈ [n] do
    ret_i ← extract H
end
return ret.
```

Adding Elements Into The Heap. (Skip if there is no time).

input: Heap $H_1, H_2, ..H_n$
```
parent ← ⌊i/2⌋
if parent > 0 and H_parent ≤ H_i then
    H_i ↔ H_parent
    Heapify-up(parent)
end
```

input: Heap $H_1, H_2, ..H_n$
```
H_n ← v
Heapify-up(n)
n ← n + 1
```

Example, Median Heap

Task: Write a datastructure that support insertion and deltion at $O(\log n)$ time and in addition enable to extract the median in $O(\log n)$ time.

Soultlion. We will define two separate Heaps, the first will be a maximum heap and store the first $\lfloor n/2 \rfloor$ smallest elements, and the second will be a minimum heap and contain the $\lceil n/2 \rceil$ greatest elements. So, it's clear that the root of the maximum heap is the median of the elements. Therefore to guarantee correctness, we should maintain the balance between the heap's size. Namely, promising that after each insertion or extraction, the difference between the heap's size is either 0 or 1.

input: Array $H_1, H_2, ..H_n, v$

if $H_{\max,1} \leq v \leq H_{\min,1}$ **then**
    if *size* $(H_{\max})-$ *size*$(H_{\min}) = 1$ **then**
        Insert-key (H_{min}, v)
    end
    else
        Insert-key (H_{max}, v)
    end
end
else
    median $\leftarrow$ Median-Extract H
    if $v <$ *median* **then**
        Insert-key (H_{max}, v)
        Insert-key ($H_{min}, median$)
    end
    else
        Insert-key (H_{min}, v)
        Insert-key ($H_{max}, median$)
    end
end

input: Array $H_1, H_2, ..H_n$

median $\leftarrow$ extract H_{max}
if *size(H_{min}) - size(H_{max})* > 0 **then**
    temp $\leftarrow$ extract $H_{\min}$
    Insert-key (H_{max}, temp)
end
return median

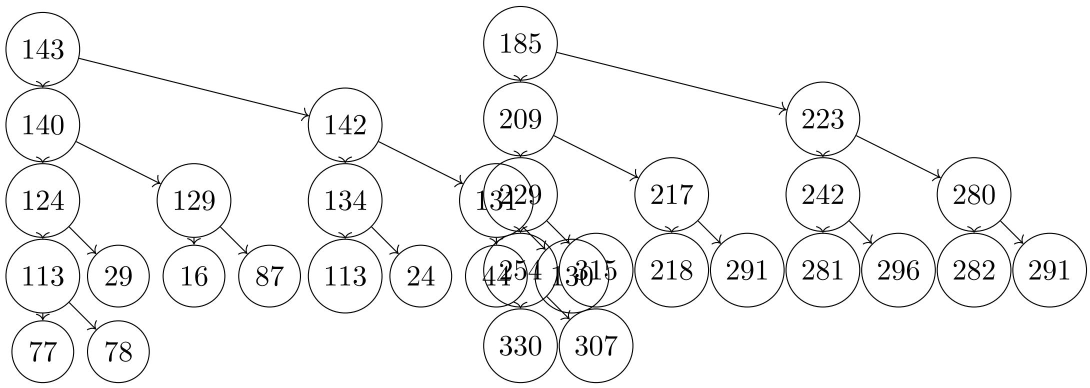


Figure 8.3: Example for Median-Heap, the left and right trees are maximum and minimum heaps.

Chapter 9

Quicksort And Liner Time Sorts - Recitation 6

Till now, we have quantified the algorithm performance against the worst-case scenario. And we saw that according to that measure, In the comparisons model, one can not sort in less than $\Theta(n \log n)$ time. In this recitation, we present two new main concepts that, in some instances, achieve better analyses. The first one is the Exception Complexity; By Letting the algorithm behave non-determinately, we might obtain an algorithm that most of the time runs to compute the task fast. Yet we will not success get down beneath the $\Theta(n \log n)$ lower bound, but we will go back to use that concept in the pending of the course. The second concept is to restrict ourselves to dealing only with particular inputs. For example, We will see that if we suppose that the given array contains only integers in a bounded domain, then we can sort it in linear time.

Quicksort.

The quicksort algorithm is a good example of a **non-determistic** algorithm that has a worst-case running time of $\Theta\left(n^2\right)$. Yet its expected running time is $\Theta(n \log n)$. Namely, fix an array of n numbers. The running of Quicksort over that array might be different. Each of them is a different event in probability space, and the algorithm's running time is a random variable defined over that space. Saying that the algorithm has the worst space complexity of $\Theta(n^2)$ means that there exists an event in which it runs $\Theta\left(n^2\right)$ time with non-zero probability. But practically, the interesting question is not the existence of such an event but how likely it would happen. It turns out that the expectation of the running time is $\Theta(n \log n)$.

What is the exact reason that happens? By giving up on the algorithm behavior century, we will turn the task of engineering a bad input impossible.

```
i ← random (p, r)
A_r ↔ A_i
return Partition (A, p, r)
```

if $p < r$ **then**
    $q \leftarrow$ randomized-partition (A, p, r)
    randomized-quicksort $(A, p, q - 1)$
    randomized-quicksort $(A, q + 1, r)$
end

Partitioning. To complete the correctness proof of the algorithm (most of it passed in the Lecture), we have to prove that the partition method is indeed re-arranging the array such that all the elements contained in the right subarray are greater than all the elements on the left subarray.

1 $x \leftarrow A_r$
2 $i \leftarrow p - 1$
3 **for** $j \in [p, r - 1]$ **do**
4   **if** $A_j \leq x$ **then**
5         $i \leftarrow i + 1$
6         $A_i \leftrightarrow A_j$
7   **end**
8 **end**
9 $A_{i+1} \leftrightarrow A_r$
10 return $i + 1$

claim. At the beginning of each iteration of the loop of lines 3–6, for any array index k, the following conditions hold:

- if $p \leq k \leq i$, then $A_k \leq x$.
- if $i + 1 \leq k \leq j$–1, then $A_k > x$.
- if $k = r$, then $A_k = x$.

Proof.

1. Initialization: Prior to the first iteration of the loop, we have $i = p$–1 and $j = p$. Because no values lie between p and i and no values lie between $i+1$ and j–1, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

2. Maintenance: we consider two cases, depending on the outcome of the test in line 4, when $A_j > x$: the only action in the loop is to increment j. After j has been incremented, the second condition holds for $A_{j–1}$, and all other entries remain unchanged. When $A_j \leq x$: the loop increments i, swaps A_i and A_j, and then increments j. Because of the swap, we now have that $A_i \leq x$, and condition 1 is satisfied. Similarly, we also have that $A_{j–1} > x$, since the item that was swapped into $A_{j–1}$ is, by the loop invariant, greater than x.

3. Termination: Since the loop makes exactly r–p iterations, it terminates, whereupon$j = r$. At that point, the unexamined subarray $A_j, A_{j+1}..A_{r–1}$ is empty, and every entry in the array belongs to one of the other three sets

described by the invariant. Thus, the values in the array have been partitioned into three sets: those less than or equal to x (the low side), those greater than x (the high side), and a singleton set containing x (the pivot).

1	5	8	5	3	2	1		1	1	2	3	5	8	5	
1	5	8	5	3	2	1	quicksort$(A, 0, 6)$	1	1	2	3	5	8	5	quicksort$(A, 4, 6)$
1	5	8	5	3	2	1		1	1	2	3	5	5	8	
1	5	8	5	3	2	1		1	1	2	3	5	5	8	
1	1	8	5	3	2	5		1	1	2	3	5	5	8	
1	1	8	5	3	2	5	quicksort$(A, 2, 6)$	1	1	2	3	5	5	8	quicksort$(A, 4, 5)$
1	1	8	5	5	2	3		1	1	2	3	5	5	8	
1	1	2	5	5	8	3		1	1	2	3	5	5	8	

Linear Time Sorts

Counting sort. Counting sort assumes that each of the n input elements is an integer at the size at most k. It runs in $\Theta(n+k)$ time, so that when $k = O(n)$, counting sort runs in $\Theta(n)$ time. Counting sort first determines, for each input element x, the number of elements less than or equal to x. It then uses this information to place element x directly into its position in the output array. For example, if 17 elements are less than or equal to x, then x belongs in output position 17. We must modify this scheme slightly to handle the situation in which several elements have the **same value**, since we do not want them all to end up in the same position.

let B and C be new arrays at size n and k
for $i \in [0, k]$ **do**
  $C_i \leftarrow 0$
end
for $j \leftarrow [1, n]$ **do**
  $C_{Aj} \leftarrow C_{A_j} + 1$
end
for $i \in [1, k]$ **do**
  $C_i \leftarrow C_i + C_{i-1}$
end
for $i \in [n, 1]$ **do**
  $B_{C_{A_j}} \leftarrow A_j$
  $C_{A_j} \leftarrow C_{Aj}{-}1$ // to handle duplicate values
end
return B

Notice that the Counting sort can beat the lower bound of $\Omega(n \log n)$ only because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the elements to index into an array.

An important property of the counting sort is that it is **stable**.

Stable Sort.

We will say that a sorting algorithm is stable if elements with the same value appear in the output array in the same order as they do in the input array.

Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

Radix sort Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column, a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

The Radix-sort procedure assumes that each element in the array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

for $i \in [1, d]$ **do**
  use a stable sort to sort array A on digit i
end

Correctness Proof. By induction on the column being sorted.

- Base. Where $d = 1$, the correctness follows immediately from the correctness of our base sort subroutine.
- Induction Assumption. Assume that Radix-sort is correct for any array of numbers containing at most $d - 1$ digits.
- Step. Let A' be the algorithm output. Consider $x, y \in A$. Assume without losing generality that $x > y$. Denote by x_d, y_d their d-digit and by $x_{/d}, y_{/d}$ the numbers obtained by taking only the first $d - 1$ digits of x, y. Separate in two cases:
  - If $x_d > y_d$ then a scenario in which x appear prior to y is imply contradiction to the correctness of our subroutine.
  - So consider the case in which $x_d = y_d$. In that case, it must hold that $x_{/d} > y_{/d}$. Then the appearance of x prior to y either contradicts the assumption that the base algorithm we have used is stable or that x appears before y at the end of the $d - 1$ iteration. Which contradicts the induction assumption.

The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit lies in the range 0 to k–1 (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes $\Theta(n+k)$ time. There are d passes, and so the total time for radix sort is $\Theta\left(d(n+k)\right)$.

Bucket sort. Bucket sort divides the interval [0, 1) into n equal-sized subintervals, or buckets, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over [0, 1), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

let B[0 : n − 1] be a new array **for** $i \leftarrow [0, n–1]$ **do**
    make B_i an empty list
end
for $i \leftarrow [1, n]$ **do**
    insert A_i into list $B_{\lfloor nA_i \rfloor}]$
end
for $i \leftarrow [0, n–1]$ **do**
    sort list B_i
end
concatenate the lists $B_0, B1, .., B_{n–1}$ together and
return the concatenated lists

Chapter 10

AVL Trees - Recitation 7

In this recitation, we will review the new data structures you have seen - Binary search trees and, specifically, AVL trees. We will revise the different operations, review the essential concepts of balance factor and rotations and see some examples. Finally, if there is time left - we will prove that the height of an AVL tree is $O(\log(n))$.

10.1 AVL trees

Reminders:

Binary Tree.

A binary tree is a tree (T, r) with $r \in V$, such that $\deg(v) \leq 2$ for any $v \in V$.

Height.

A tree's height $h(T)$ (sometimes $h(r)$) is defined to be the length of the longest simple path from r to a leaf.

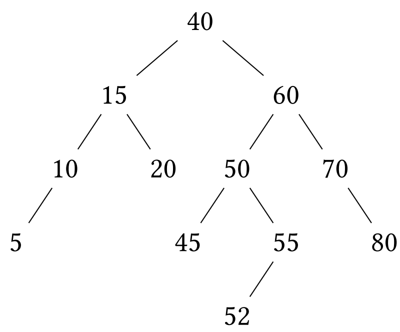
Binary Search Tree.

A binary search tree is a binary tree (T, r) such that for any node x (root of a subtree) and a node in that subtree y:

1. if y is in the left subtree of x then $y.key \leq x.key$
2. if y is in the right subtree of x then $x.key < y.key$

Note that this is a (relatively) local property.

For example:



Remark 10.1.1. Go over the properties and calculate the tree's height. Make sure you understand the definitions!

Last time, we saw some operations that can be performed on BSTs, and proved correctness for some of them. These were: $Search(x), Min(T), Max(T), Pred(x), Succ(x), Insert(x$
All of these operations take $O(h(T))$ in the worst case.

The main two operations that may cause problems are $Insert$ and $Delete$, as they change the tree's height (consider inserting $81, 82, 83, 84$ to our working example). To address this problem, we introduce another field: for each node v, add a field of $h(v)$ = the height of the subtree whose root is v. This allows us to maintain the AVL property:

AVL Tree.

An AVL tree is a balanced BST, such that for any node x, its left and right subtrees' height differs in no more than 1. In other words:

$$|h(left(x)) - h(right(x))| \leq 1$$

This field allows us to calculate the Balance Factor for each node in $O(1)$:

Balance Factor.

For each node $x \in T$, it's Balance Factor is defined

$$hd(x) := h(left(x)) - h(right(x))$$

In AVL trees, we would like to maintain $|hd(x)| \leq 1$

Example 10.1.1. For our working example, the node 60's hd is $h(50) - h(70) = 1$, and $hd(50) = h(45) - h(55) = -1$. You can check and see that this is an AVL tree.

So to make sure that we can actually maintain time complexity $O(\log(n))$, we would want to:

1. Show that for an AVL tree, $h(T) = \theta(log(n))$ (If there's time left)
2. See how to correct violations in AVL property - using **rotations**
3. See how to $Delete$ and $Insert$, while maintaining the height field.

10.1.1 Rotations

Recitations/rotations.png

Rotations allow us to maintain the AVL property in $O(1)$ time (you have discussed this in the lecture - changing subtree's roots). In this schematic representation of rotations, x, y are nodes and α, β, γ are subtrees. Note that the BST property is maintained!

The Balance factor allows us to identify if the AVL property was violated, and moreover - the exact values of the bad Balance factors will tell us which rotations to do to fix this:
Taken from last year's recitation:

Recitations/violations.png

Let us analyze one of these cases:

Example 10.1.2. Let us see why R rotation Fixes LL violation:

Recitations/LLviolation.jpg

This is the general form of LL violations.

Let us analyze the heights of subtrees. Denote h the height of A_L before inserting v. So A_R's height has also to be h. If its height were $h+1$, the insertion would not have created a violation in B (A's height would have stayed the same). If it were $h-1$, the violation would have appeared first in A (not in B). Thus, A's height is $h+1$. B_R's height is also h: If it was $h+1$ or $h+2$, no violation in B had occurred.

After the rotation, the tree looks like this: So all the nodes here maintain AVL

Recitations/LLviolationfix.jpg

property; why is it maintained?

Detect the violation in the following tree, and perform the necessary rotations:

10.1.2 Delete, Insert.

The principles of the $Delete$ and $Insert$ operations are the same as in regular BST, but we will need to rebalance the tree in order to preserve AVL property.

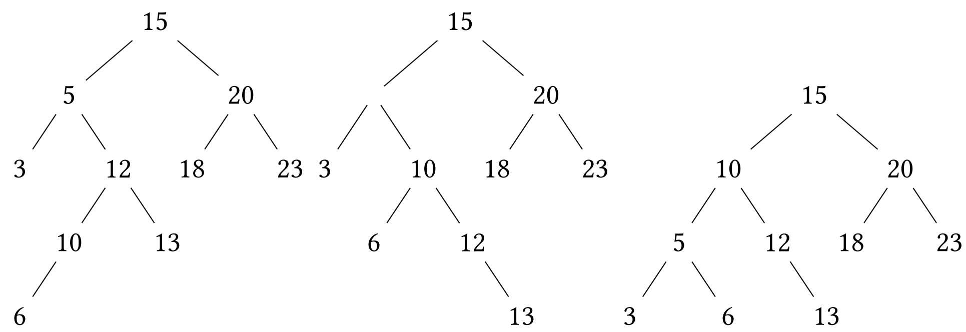


Figure 10.1: The first node in which a violation occurred is 5, this is an RL violation. Perform R rotation on the right child. Then perform L rotation on the root of the relevant subtree (5):

A single insertion or deletion may change the height difference of subtrees by at most 1, and might affect only the subtrees with roots along the path from r to the point of insertion/ deletion. More concretely - we will add a recursive operation of traversing the tree "back up" and checking violations. Had we found one - we will fix it using rotations. Since rotations can be done in $O(1)$, the entire correction process will take $O(\log(n))$, so we maintain a good time complexity.

Call Insert (r, x) // (The standard insertion routine for BST)
Let p be the new node appended to the tree.
while $p.parent \neq \emptyset$ **do**
    Check which of the four states we are in.
    Apply the right rotation.
    Update the height of each touched vertex
    $p \leftarrow p$.parent
end

10.1.3 Exam Question.

Consider the following question from 2018 exam.

אתם מתבקשים לממש מבנה נתונים התומך בפעולות הבאות:

insert(x): הכנסת המספר x למבנה, אם x לא נמצא בו כבר.

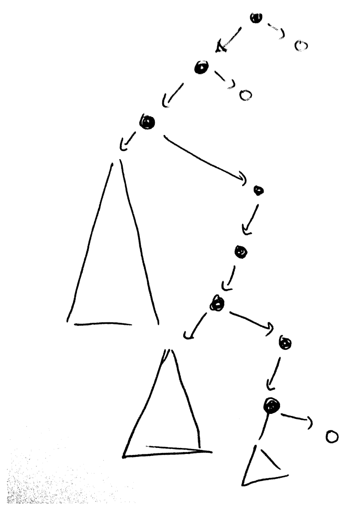
delete(x): מחיקת x מהמבנה, אם x נמצא בו.

sum_over(x): מחזיר את סכום כל המספרים במבנה שערכם גדול או שווה לזה של x.

סיבוכיות כל פעולה צריכה להיות O(log n), כאשר n הוא מספר האיברים הנוכחי במבנה.

Solution. How would it look like an algorithm which computes the 'sum over' query? Or for given x, which vertices have a key less than x? Consider the path $v_1, v_2, v_3 \ldots v_n$ on which the Search subroutine went over when looking for x. If v_i.key $< v_{i+1}$.key then it means that x is greater then v_i.key. Furthermore, x is also greater than all the keys in the left subtree of v_i.

Let us transform that insight into an Algorithm. Define for each vertex a field that stores the summation of all its left subtree keys plus its own key; call it 'Leftsum'. Then computing the sumover query will be done by summing these fields of the vertices in the right turn on the searching path.



```
if r is None then
|   return 0
end
else
|   if x > r.key then
|   |   return r.Leftsum + Sumover( r.right, x )
|   end
|   else
|   |   return Sumover( r.left, x )
|   end
end
```

So it has left to show how one could maintain the tree and guarantee a logarithmic height. Consider a vertex v and suppose that both his direct children are correct AVL trees and have the correct value at the Leftsum field. We know that:

1. There is a rotation that balances the tree at the cost of $O(1)$ time.

2. All the descendants of v, at a distance at least two from it, will remain the same in the sense that their subtree has not changed.

Therefore we will have to recompute the Sumleft filed for only a $O(1)$ vertices (which are the children and the grandchildren of v previews the rotation). Each computation could be done in $O(1)$ time.

Call Insert (r, x) // (The standard insertion routine for BST)
Let p be the new node appended to the tree.
while $p.parent \neq \emptyset$ **do**
  Apply the right rotation.
  for *any vertex v such that its pointers have changed, sorted by height* **do**
    v.Leftsum $\leftarrow$ v.key + v.left.Leftsum + v.left.right.Leftism
  end
  $p \leftarrow p$.parent
end

Running time. The work is done over vertices along a single path, and on each vertex only $O(1)$ time is spent, Then we have that the total running time of the algorithm is proportional to the tree height. Combining the fact that we maintain the AVL property, it follows that the total time is $\Theta(\log n)$.

Appendix

10.1.4 AVL tree's height

Let n_h be the minimal number of nodes in an AVL tree of height h.
n_h is strictly increasing in h.

Proof. Exercise. □

$$n_h = n_{h-1} + n_{h-2} + 1.$$

Proof. For an AVL tree of height 0, $n_0 = 1$, and of height 1, $n_1 = 2$ (by checking directly).
Let's look at a minimal AVL tree of height h. By the AVL property, one of its subtrees is of height $h-1$ (WLOG - the left subtree) and by minimality, its left subtree has to have n_{h-1} nodes. T's right subtree thus has to be of height $h-2$: It can't be of height $h-1$: $n_{h-1} > n_{h-2}$ by the previous theorem, and if the right subtree is of height $h-1$ - we could switch it with an AVL tree of height $h-2$, with fewer nodes - so less nodes in T, contradicting minimality. So the right subtree has n_{h-2} nodes (once again, by minimality), and thus the whole tree has $n_h = n_{h-1} + n_{h-2} + 1$ (added the root) nodes. □

$$n_h > 2n_{h-2} + 1 \quad h = O(\log(n))$$

Proof. Assume k is even (why can we assume that?). It can be shown by induction that:

$$n_h > 2n_{h-2}+1 > 2(2n_{h-4}+1)+1 = 4n_{h-4}+(1+2)\ldots > 2^{\frac{h}{2}}+\sum_{i=0}^{\frac{h}{2}-1} 2^i = \sum_{i=1}^{\frac{h}{2}} 2^i = \frac{2^{\frac{h}{2}}-1}{2-1} = 2^{\frac{h}{2}}-1$$

So $n_h \geq 2^{\frac{h}{2}} - 1$, thus
$$h \leq 2\log(n_h + 1)$$

and for generall AVL tree with n nodes and height h:

$$h \leq 2\log(n_h + 1) \leq 2\log(n + 1) = O(\log(n))$$

$\square$

Remark 10.1.2. In fact, one can show that $n_h > F_h$ and F_h is the h'th Fibonacci number. Recall that $F_h = C(\varphi^h - (\psi)^h)$, and this gives a tighter bound on n_h.

Chapter 11

Graphs - Recitation 9

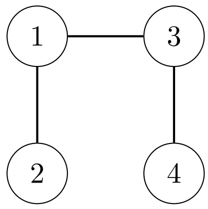
11.1 Graphs

This is an important section, as you'll see graphs A LOT in this course and in the courses to follow.

11.1.1 Definitions, Examples, and Basics

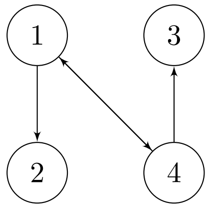
Definition 11.1.1. A **non-directed graph** G is a pair of two sets - $G = (V, E)$ - V being a set of vertices and E being a set of couples of vertices, which represent edges ("links") between vertices.

Example: $G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{3, 4\}\})$ is the following graph:



Definition 11.1.2. A **directed graph** G is a pair of two sets - $G = (V, E)$ - V being a set of vertices and $E \subseteq V \times V$ being a set of directed edges ("arrows") between vertices.

Example: $G = (\{1, 2, 3, 4\}, \{(1, 2), (1, 4), (4, 1), (4, 3)\})$ is the following graph (note that it has arrows):



Definition 11.1.3. A **weighted graph** composed by a graph $G = (V, E)$ (either non-directed or directed) and a weight function $w : E \to \mathbb{R}$. Usually (but not necessary), we will think about the quantity $w(e)$, where $e \in E$, as the length of the edge.

Now that we see graphs with our eyes, we can imagine all sorts of uses for them... For example, they can represent the structure of the connections between friends on Facebook, or they can even represent which rooms in your house have doors between them.

Remark 11.1.1. Note that directed graphs are a **generalization** of non-directed graphs, in the sense that every non-directed graph can be represented as a directed graph. Simply take every non-directed edge $\{v, u\}$ and turn it into two directed edges $(v, u), (u, v)$.

Remark 11.1.2. Note that most of the data structures we discussed so far - Stack, Queue, Heap, BST - can all be implemented using graphs.

Now let's define some things in graphs:

Definition 11.1.4. (Path, circle, degree)

1. A **simple path** in the graph G is a series of unique vertices (that is, no vertex appears twice in the series) $v_1, v_2, ..., v_n$ that are connected with edges in that order.

2. A **simple circle** in the graph G is a simple path such that $v_1 = v_n$.

3. The **distance** between two vertices $v, u \in V$ is the length of the shortest path between them (∞ if there is no such path).

Remark 11.1.3. Note that for all $u, v, w \in V$ the triangle inequality holds regarding path lengths. That is:

$$dist(u, w) \leq dist(u, v) + dist(v, w)$$

Definition 11.1.5. (connectivity)

1. Let $G = (V, E)$ be a non-directed graph. A **connected component** of G is a subset $U \subseteq V$ of maximal size in which there exists a path between every two vertices.

2. A non-directed graph G is said to be a **connected** graph if it only has one connected component.

3. Let $G = (V, E)$ be a directed graph. A **strongly connected component** of G is a subset $U \subseteq V$ of maximal size in which for any pair of vertices $u, v \in U$ there exist both directed path from u to v and a directed path form v to u.

Let $G = (V, E)$ be some graph. If G is connected, then $|E| \geq |V| - 1$

Proof. We will perform the following cool process: Let $\{e_1, ..., e_m\}$ be an enumeration of E, and let $G_0 = (V, \emptyset)$. We will build the graphs $G_1, G_2, ...G_m = G$ by adding edges one by one. Formally, we define -

$$\forall i \in [m] \ \ G_i = (V, \{e_1, ..., e_i\})$$

G_0 has exactly $|V|$ connected components, as it has no edges at all. Then G_1 has $|V| - 1$. From there on, any edges do one of the following:

1. Keeps the number of connected components the same (the edge closes a cycle) '

2. Lowers the number of connected components by 1 (the edges does not close a cycle)

So in general, the number of connected components of G_i is $\geq |V| - i$. Now, if $G_m = G$ is connected, it has just one connected component! This means:

$$1 \geq |V| - |E| \implies |E| \geq |V| - 1$$

□

11.1.2 Graph Representation

Okay, so now we know what graphs are. But how can we represent them in a computer? There are two main options. The first one is by **array of adjacency lists**. Given some graph G, every slot in the array will contain a linked list. Each linked list will represent a list of some node's neighbors. The second option is to store edges in an **adjacency matrix**, a $|V| \times |V|$ binary matrix in which the v, u-cell equals 1 if there is an edge connecting u to v. That matrix is denoted by A_G in the example below. Note that the running time analysis might depend on the underline representation.

Question. What is the memory cost of each of the representations? Note that while holding an adjacency matrix requires storing $|V|^2$ bits regardless of the size of E, Maintaining the edges by adjacency lists costs linear memory at the number of edges and, therefore, only $\Theta(|V| + |E|)$ bits.

Example. Consider the following directed graph:

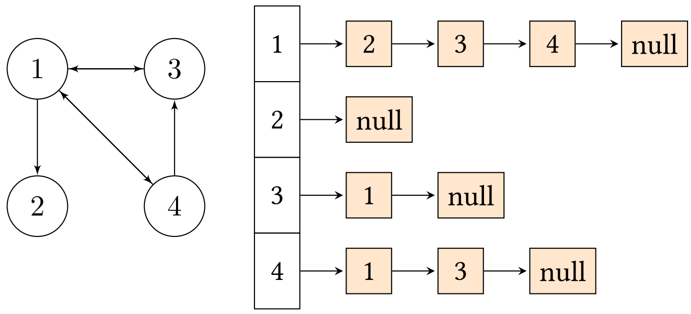
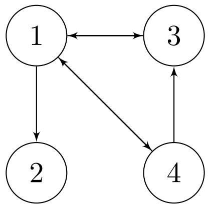


Figure 11.3: Persenting G by array of adjacency lists.

11.1.3 Breadth First Search (BFS)

One natural thing we might want to do is to travel around inside a graph. That is, we would like to visit all of the vertices in a graph in some order that relates to the edges. Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever the search discovers a white vertex v in the course of scanning the adjacency list of a gray vertex u, the vertex v and the edge (u, v) are added to the tree. We say that u is the predecessor or parent



$$A_G = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 11.4: Presenting G by adjacency matrix.

of v in the breadth-first tree. Since every vertex reachable from s is discovered at most once, each vertex reachable from s has exactly one parent. (There is one exception: because s is the root of the breadth-first tree, it has no parent.) Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on the simple path in the tree from the root s to vertex v, then u is an ancestor of v, and v is a descendant of u. The breadth-first-search procedure BFS on the following page assumes that the graph $G = (V, E)$ is represented using adjacency lists. It denotes the queue by Q, and it attaches three additional attributes to each vertex v in the graph:

1. v.visited is a boolean flag which indicate wheter v was allready visited.

2. $\pi(v)$ is v's predecessor in the breadth-first tree. If v has no predecessor because it is the source vertex or is undiscovered, then $\pi(v)$ is None/NULL.

```
1  for v ∈ V do
2  |  v.visited ← False
3  end
4  Q ← new Queue
5  Q.Enqueue(s)
6  s.visited ← True
7  while Q is not empty do
8  |  u ← Q.Dequeue()
9  |  for neighbor w of u do
10 |  |  if w.visited is False then
11 |  |  |  w.visited← True
12 |  |  |  π(w) ← u
13 |  |  |  Q.Enqueue(w)
14 |  |  end
15 |  end
16 end
```

Correctness: The example should be enough to explain the correctness. A concrete proof can be found in the book, page 597.

Runtime: We can analyse the runtime line-by-line:

- Lines 1-2: $|V|$ operations, all in $O(1)$ runtime, for a total of $O(|V|)$.

- Lines 3-6: $O(1)$

- Lines 7-8: First we need to understand the number of times the $while$ loop iterates. We can see that every vertex can only enter the queue ONCE (since it is then tagged as "visited"), and therefore it runs $\leq |V|$ times. All operations are $O(1)$, and we get a total of $O(|V|)$.

- Lines 9-13: Next, we want to understand the number of times this for loop iterates. The for loop starts iterating once per vertex, and then the number of its iterations is the same as the number of neighbors that this vertex has. Thus, it runs $O(|E|)$ times.

So all in all we get a runtime of $O(|V| + |E|)$

11.1.4 Usage of BFS

Now we have a way to travel through a graph using the edges. How else can we use it?

Exercise: Present and analyse an algorithm $CC(G)$ which receives some undirected graph G and outputs the number of connected components in G in $O(|V|+|E|)$.

Solution: Consider the following algorithm: (Now read the algorithm, it may be in the next page because LaTeX is dumb...)

```
count ← 0
for v ∈ V do
    if v.visited = False then
        count←count+1
        BFS(G, v)
    end
end
return count
```

11.1.5 Depth First Search (DFS)

As its name implies, depth-first search searches "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until all vertices that are reachable from the original source vertex have been discovered. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, repeating the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

```
DFS( G ):
for v ∈ V do
    vi.visited ← False
end
time ← 1
for v ∈ V do
    if not v.visited then
        π (v) ← null
        Explore( G, v )
    end
end
```

```
Explore(G, v):
Previsit(v) for (v, u) ∈ E do
    if not u.visited then
        π (u) ← v
        Explore( G, u )
    end
end
Postvisit(v)
```

```
Previsit(v):
pre(v) ← time
time ← time +1
```

```
Postvisit (v):
post(v) ← time
time ← time +1
```

Properties of depth-first search. Depth-first search yields valuable information about the structure of a graph. Perhaps the most basic property of depth-first search is that the predecessor subgraph G_π does indeed form a forest of trees since the structure of the depth-first trees exactly mirrors the structure of recursive calls of explore-function. That is, $u = \pi(v)$ if and only if explore(G, v) was called during a search of u's adjacency list. Additionally, vertex v is a descendant of vertex u in the depth-first forest if and only if v is discovered during the time in which u is gray. Another important property of depth-first search is that discovery and finish times have a parenthesis structure. If the explore procedure were to print a left parenthesis "$(u$" when it discovers vertex u and to print a right parenthesis r"$u)$" when it finishes u, then the printed expression would be well-formed in the sense that the parentheses are properly nested.

The following theorem provides another way to characterize the parenthesis structure.

Parenthesis theorem In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

1. the intervals [pre(u), post(u)] and [pre(v), post(v)] are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest.

2. the interval [pre(u), post(u)] is contained entirely within the interval [pre(v), post(v)], and u is a descendant of v in a depth-first tree, or

3. the interval [pre(v), post(v)] is contained entirely within the interval [pre(u), post(u)], and v is a descendant of u in a depth-first tree.

Proof. We begin with the case in which pre(u) $<$ pre(v). We consider two subcases, according to whether pre(v) $<$ post(u). The first subcase occurs when pre(v) $<$ post(u), so that v was discovered while u was still gray, which implies that v is a descendant of u. Moreover, since v was discovered after u, all of its outgoing edges are explored, and v is finished before the search returns to and finishes u. In this case, therefore, the interval [pre(v), post(v)] is entirely contained within the interval [pre(u), post(u)]. In the other subcase, post(u) $<$ pre(v), and by defintion, pre(u) $<$ post(u) $<$ pre(v) $<$ post(v), and thus the intervals [pre(u), post(u)] and [pre(v), post(v)] are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.

Corollary. Nesting of descendants' intervals. Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if pre(u) $<$ pre(v) $<$ post(v) $<$ post(u).

Chapter 11

Minimum Spanning Tree Recitation.

11.1 The MST Problem.

Definition 11.1.1. A spanning tree T of a graph $G = (V, E)$ is a subset of edges in E such that T is a tree (having no cycles), and the graph (V, T) is connected.

Problem 11.1.1 (MST). *Let $G = (V, E)$ be a weighted graph with weight function $w : E \to \mathbb{R}$. We extend the weight function to subsets of E by defining the weight of $X \subset E$ to be $w(X) = \sum_{e \in X} w(e)$. The minimum spanning tree (MST) of G is the spanning tree of G that has the minimal weight according to w. Note that in general, there might be more than one MST for G.*

Definition 11.1.2. Let $U \subset V$. We define the cut associated with U as the set of outer edges of U, namely all the edges $(u, v) \in E$ such that $u \in U$ and $v \notin U$. We use the notation $X = (U, \bar{U})$ to represent the cut. We say that $E' \subset E$ respects the cut if $E' \cap X = \emptyset$.

Lemma 11.1.1 (The Cut-Lemma). *Let T be an MST of G. Consider a forest $F \subset T$ and a cut X that respects X (i.e. $F \cap X = \emptyset$). Then $F \cup \ arg\min_e w(e)$ is also contained in some MST. Note that it does not necessarily have to be the same tree T.*

Proof. Separate to cases:

- If $e \in T$, then $F \cup \{e\} \subset T$ and we done.
- Otherwise, consider the second case where $e \notin T$. This means that $T \cup \{e\}$ has $|V|$ edges and therefore must have a cycle. Let $\Gamma = T \cup \{e\}$ and let x and y be the endpoints of e (namely $e = (x, y)$). Denote the subset of vertices defining the cut X by U. Without loss of generality, let's assume $x \in U$ and $y \in \bar{U}$.

  Since T is connected, there is a path $x \rightsquigarrow y$ in T, denote it by $\mathcal{P}$. Additionally, because $e \notin T$, we have that $e \notin \mathcal{P}$. This means that there must be another edge in $\mathcal{P}$ connecting a vertex in U to a vertex in $\bar{U}$[1]. Let e' be that edge, we have:

[1] Otherwise, walking along $\mathcal{P}$ cannot take one out of U, leading to a contradiction as $\mathcal{P}$ leads to y.

1. Both $e', e \in X$ So $w(e) \leq w(e')$.
2. $e \cup \mathcal{P}$ is a cycle in Γ.

By using the fact that subtracting an edge from a cycle doesn't harm connectivity (see Claim 11.3.1), we can conclude that $\Gamma/\{e'\}$ is connected. Since it has $|V| - 1$ edges, it must be a spanning tree. On the other hand, by:

$$w\left(\Gamma/\{e'\}\right) = w\left(T\right) + \overbrace{w(e) - w(e')}^{\leq 0} \leq w\left(T\right)$$

So $\Gamma/\{e'\}$ is an MST. Finally, to close the proof, observe that $F \cup \{e\} \subset \Gamma/\{e'\}$. This means that, we have found an MST that contains $F \cup \{e\}$.

□

11.2 Kruskal Algorithm.

This algorithm constructs the MST iteratively by holding a forest F contained in an MST and then looking for the minimal edge in a cut that it respects. Note, that since F has no cycles, any edge $e \subset E$ that does not create a cycle in F must belong to a cut X that is respected by F.

By ensuring that the edges are examined in increasing weight order, we can determine that the first edge that does not create a cycle is also the one with the minimum weight among them. Therefore, according to Lemma 11.1.1, we can conclude that the forest obtained by adding e into F is contained in an MST.

Result: Returns MST of given $G = (V, E, w)$
1 sorts E according to w
2 define $F_0 = \emptyset$ and $i \leftarrow 0$
3 **for** $e \in E$ *in sorted order* **do**
4   **if** $F_i \cup \{e\}$ *has no cycle* **then**
5         $F_{i+1} \leftarrow F_i \cup \{e\}$
6         $i \leftarrow i + 1$
7   **end**
8 **end**
9 **return** F_i

Algorithm 26: Kruskal alg.

Claim 11.2.1. *For any i, $F_i \subset$ of an MST.*

Proof. By induction.

1. Base. Let T be an arbitrary MST of G. $F_0 = \emptyset \subset T$.

2. Assumption. Assume correctness for any $j < i$.

3. Step. By the induction assumption, there is an MST T such that $F_{i-1} \subset T$. Denote by $e = (x, y)$ the edge for which $F_i = F_{i-1} \cup \{e\}$. According to the algorithm definition, $F_i = F_{i-1} \cup \{e\}$ has no cycles (line number (4)). This

means that with respect to F_{i-1}, x and y belong to two different connected components. Denote the connected component of x by U, and the cut it defines by $X = (U, \bar{U})$. It is clear that F_{i-1} respects X (otherwise U would not be a connected component of F_{i-1}).

We calim that e is the minaiml edge in X. Any other e' with $w(e') < w(e)$ is either already in F_{i-1} and therefore cannot be in X, or it closes a cycle in F_j for some $j < i$. Since $F_j \subset F_{i-1}$, it also closes a cycle in F_{i-1}. Therefore, it cannot be an edge connecting between U and $\bar{U}$, namely $e' \notin X$.

So, if F_{i-1} respects X and e is the minimal edge in X, then it follows from Lemma 11.1.1 that $F_i = F_{i-1} \cup \{e\}$ is contained in an MST.

□

Problem 11.2.1. *Let $E' \subset E$ such E contains both an MST T and a cycle C. Let e be a maximal edge in C. Prove that $E'/\{e\}$ contains an MST.*

Solution 11.2.1. Separate to cases:

- If $e \notin T$, then we done.
- So, it is left to prove for $e \in T$. Let $(x, y) = e$ and consider the forest $F = T/\{e\}$.

  Since T is a spanning tree, subtracting e from T divides T into two connected components, U and $\bar{U}$, corresponding to all vertices that can be reached from x and y, respectively. Let X be the cut $X = (U, \bar{U})$. Note that F respects X. On the other hand, since (x, y) is an edge in cycle C, there is another path from x to y that does not contain e. This path must have a non-trivial intersection with X (otherwise, walking through the path cannot lead to a vertex in $\bar{U}$).

  Therefore, there exists an edge $e' \neq e$ such that $e \in C \cap X$. Let $e' = (u, v)$ and assume, without loss of generality, that $u \in U$ and $v \in \bar{U}$. Since U and $\bar{U}$ are connected components, there are paths $x \rightsquigarrow u$ and $v \rightsquigarrow y$ that connect with e and e'. This creates a cycle in $T \cup \{e'\}$. Using the fact that subtracting an edge from a cycle does not harm the graph's connectivity, it follows that $T' = T \cup \{e'\}/\{e\}$ is connected and therefore a spanning tree as well.

  Furthermore, $w(T') = w(T) - w(e) + w(e') \leq w(T)$. Finally, observe that $T' \subset E'/\{e\}$, and we get the required result.

Problem 11.2.2. *Consider Problem 11.2.1 and the it's solution, give an example to a graph G and subset of edges E' such that e' defined in the solution is not the minimal edge in C.*

Problem 11.2.3. *Give an example to a graph with unique MST in which the second spanning tree, share no edge with the MST.*

11.3 Appendix.

Claim 11.3.1. *Let G be a connected graph containing a cycle C. Then the subtraction of any edge in C gives a connected graph.*

Proof. Assume, by contradiction, that a graph $G' = G/\{e\}$, where $e \in C$, is not connected. This means that there are two vertices u and v that have a path between them in G, but no such path exists in G'. Denote this path by $\mathcal{P}$ and observe that $e \in \mathcal{P}$, otherwise, $\mathcal{P}$ would also be a path from u to v in G'.

Denote the ends of e by $(x, y) = e$. Also, denote C by $\langle x_0, x_1, ..x_i, x, y, y_0, .., y_j \rangle$, where $y_j = x_0$ and there is an inequality for any other pair of vertices (we used the cycle definition). Then, there is a path $x \rightsquigarrow y$ in C, defined by

$$\langle x_i, x_{i-1}, .., x_1, x_0, y_{j-1}, y_{j-2}, .., y_0, y \rangle$$

We denote this path by $\mathcal{P}'$. By replacing e in $\mathcal{P}$ with $\mathcal{P}'$, we obtain a path $u \rightsquigarrow x \rightsquigarrow^{\mathcal{P}'} y \rightsquigarrow v$, which is a path between u and v that does not contain e. This contradicts the assumption that there is no path between u and v in G'. $\square$

Chapter 12

Union Find - Recitation 11

12.1 Union Find.

We have mentioned that to find efficiently the minimal spanning tree using Kruskal, One has to answer quickly about whether a pair of vertices v, u share the same connectivity component. In this recitation, we will present a data structure that will allow us to query the belonging of a given item and merge groups at an efficient time cost.

The problem defines as follows. Given n items $x_1 ... x_n$, we would like to maintain the partition of them into disjoints sets by supporting the following operations:

1. Make-Set(x) create an empty set whose only member is x. We could assume that this operation can be called over x only once.

2. Union(x, y) merge the set which contains x with the one which contains y.

3. Find-Set(x) returns a pointer to the set holding x.

Notice that the native implementation using pointers array, A, defined to store at place i a pointer to the set containing x can perform the Find-Set operation at $O(1)$. The bottleneck of that implementation is that the merging will require us to run over the whole items and changes their corresponding pointer at A one by one. Namely, a running time cost of $\Theta(n)$ time. Let's review a diffrent approch:

Linked Lists Implementation. One way to have a non-trivial improvement is to associate each set with a linked list storing all the elements belonging to the set. Each node of those linked lists contains, in addition to its value and sibling pointer, a pointer for the list itself (the set). Consider the merging operation again. It's clear that having those lists allow us to unify sets by iterating and updating only the elements that belong to them. Still, one more trick is needed to achieve a good running cost.

```
if size A[x] ≥ sizeA[y] then
    size A[x] ← size A[x] + size A[y]
    for z ∈ A[y] do
        A[z] ← A[x]
    end
    A[x] ← A[x] ∪ A[y] // O(1) concatenation of linked lists.
end
else
    Union (y, x)
end
```

Executing the above over sets at linear size requires at least linear time. Let's analyze what happens when merging n times. As we have seen in graphs, the runtime can be measured by counting the total number of operations each item/vertex does along the whole running. So we can ask ourselves how many times an item change its location and its set pointer. Assume that at the time when x were changed $A[x]$ contains (before the merging) t elements then immediately after that $A[x]$ will store at least $2t$ elements:

$$\text{size } A^{(t+1)}[x] \leftarrow \text{size } A^{(t)}[x] + \text{size } A^{(t)}[y] \geq 2A^{(t)}[x]$$

Hence, if we list down the sizes of the x's set at the moments merging occurred, we could write only $\log n$ numbers before exceeding the maximal size (n). That proves that the number of times the vertex changed his pointer is bounded by $\log n$, and the total number of actions costs at most $\Theta(n \log n)$.

Notice that in the case in which $m = O(1)$, we will still pay much more than needed. Anyhow the next implementation is going to give us (eventually) a much faster algorithm.

Forest Implementation. Instead of associating each set with a linked list, one might attach a first. The vertices hold the values of the items, and we could think about the root of each tree as the represtive of the tree. If two vertices x, y share the same root, then it's clear they belong to the same set. At the initialization stage, Make-Set defines the vertices as roots of trivial trees (single root without any descendants). Then the find method is:

```
while π(x) ≠ None do
    x ← π(x)
end
Return x
```

We will see that a slight change should be set for the last improvement. But before that, let's try to mimic the decision rule above. Even those, we could define a size field for each root and get the same algorithm as above. Instead, we will define another field that, from first sight, looks identical. Let the rank(v) of the node v be the height of the v. Recall that tree's height defines to be the longest path from the root to one of the vertices.

Union By Rank Huristic[1] . So as we said, first, we will ensure how to mimic the $\log n$ complexity proof under the first implementation.

```
1  x ← Find(x)
2  y ← Find(y)
3  if x ≠ y then
4      if rank(y) < rank(x) then
5          π (y) ← x
6      else if rank(y) = rank(x) then
7          π (y) ← x
8          rank(x) ← rank(x) + 1
9      else
10         π (x) ← y
11     end
12 end
```

The decision rule in lines (4-8) preserves the correctness of the following claim: Claim, let $M(r)$ a lower bound over the size of the tree at rank r. Then $M\left(r+1\right) \geq 2M\left(r\right)$. The proof is left as an exercise. Assuming the correctness of the claim, it holds that $M\left(\log n\right) \geq n$. So it immediately follows that the running time takes at most $n \log n$. We could get even tighter bound by noticing that the rank bounds a single query. And therefore, the total cost is at most $m \cdot \log n$.

Path Compression Heuristic. The final trick to yield a sub-logarithmic time algorithm is to compress the brunch on which we have already passed and reduce the number of duplicated transitions.

```
if π (x) ≠ None then
    π (x) ←Find(π (x))
end
else
    Return x
end
Return π (x)
```

Let's analyze the query cost by counting the edges on which the algorithm went over. Denote by finding $(v^{(t)})$ the query which was requested at time t and let $P^{(t)} = v, v_2..v_k$ be the vertices path on which the algorithm climbed from v up to his root. Now, observes that by compressing the path, the ranks of the vertices in P must be distinct. Now consider any partition of the line into a set of buckets (segments) $\mathcal{B} = \{B_i | B_i = [b_i, b_{i+1}]\}$.

[1]Corman calls that rule a heuristic, but please notice that heuristics usually refers to methods that seem to be efficient empirically, yet it doesn't clear how to prove their advantage mathematically. Still, in that course, we stick to Corman terminology.

$$\begin{aligned} T(n,m) &= \text{direact parent move } + \text{ climbing moves } = \\ &= \text{direact parent move } + \text{ stage exchange } + \text{ inner stage } = \\ &\le m + m\cdot|\mathcal{B}| + \sum_{B\in\mathcal{B}} \text{ steps inside B} \\ &\le m + m\cdot|\mathcal{B}| + \sum_{B\in\mathcal{B}}\sum_{rank(u)\in B} \text{ steps inside B started at } u \\ &\le m + m\cdot|\mathcal{B}| + \sum_{B\in\mathcal{B}}\sum_{rank(u)\in B} |B| \end{aligned}$$

For example, consider our last calculation, In which we divided the ranges of ranks into $\log n$ buckets at length 1, $B_r = \{r\}$, then as the size of the subtrees at rank r is at least 2^r we have that the size of $|B_r|$ is at most $\frac{n}{2^r}$ and that's why:

$$\sum_{b\in\mathcal{B}}\sum_{rank(u)\in B}|B| \le \sum_{b\in\{[i]|i\in[\log n]\}} \frac{n}{2^r}\cdot 1 \le 2\cdot n$$

So the total time is at most $m + m\log n + 2n = \Theta(n)$. And if we would take the $\log\log n$ buckets such that B_i stores the ith $\log n/\log\log n$ numbers. Then the sum above will become:

$$\begin{aligned} &\sum_{b\in\mathcal{B}}\sum_{rank(u)\in B}|B| \le \sum_{b\in\{B\in\mathcal{B}\}} \frac{n}{2^{\frac{i\log n}{\log\log n}}}\cdot\log\log n \le 2\cdot n \\ &\le n \sum_{b\in\{B\in\mathcal{B}\}} 1\cdot\cdot\log\log n \le n(\log\log n)^2 \\ &\Rightarrow m + m\cdot\log\log n + n(\log\log n)^2 \end{aligned}$$

Could we do even better? Yes, Consider a nonunifom parttion $B_r = \{r, r+1 ... 2^r\}$. So first question one should ask is, what is $|\mathcal{B}|$? ($\log^*(n)$). On the other hand, the number of vertices in which their rank belongs to the ith bucket is at most:

$$\begin{aligned} &\text{maximal number of nodes at rank } r + \text{ maximal number of nodes at rank } (r+1) + \\ &\text{maximal number of nodes at rank } (r+2) + ... + \text{ maximal number of nodes at rank } 2^r \\ &\frac{n}{2^r} + \frac{n}{2^{r+1}} + \frac{n}{2^{r+2}} + ... + \frac{n}{2^{2^r}} \Rightarrow |\{v\in B_r\}| \le 2\cdot\frac{n}{2^r} \end{aligned}$$

$$\sum_{b\in\mathcal{B}}\sum_{rank(u)\in B}|B| \le \sum_{b\in\mathcal{B}}\sum_{rank(u)\in B}\frac{2n}{2^r}|B| \le \sum_{b\in\mathcal{B}}\sum_{rank(u)\in B} 2n \le \log^{(*)}(n)\cdot 2n$$

Bibliography

[Fun21] Stanley P. Y. Fung. *Is this the simplest (and most surprising) sorting algorithm ever?* 2021. arXiv: 2110.01111 [cs.DS]. URL: https://arxiv.org/abs/2110.01111.