Discussion Midterm Exam

Algorithm Theory
WS 2013/14

Fabian Kuhn

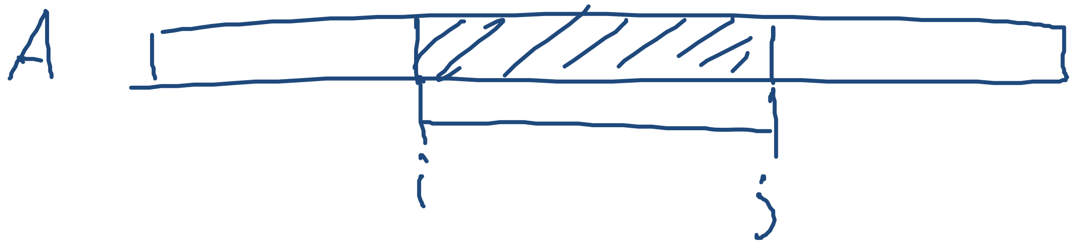
P1: Maximum Subarray Sum (18pt)

The input to the problem is an integer array A of length n. The goal is to find a subarray such that the sum of the elements in the subarray is as large as possible. Formally, the output should be two integers $1 \leq i \leq j \leq n$ such that the sum

$$\sum_{k=i}^{j} A[k]$$

is maximized.

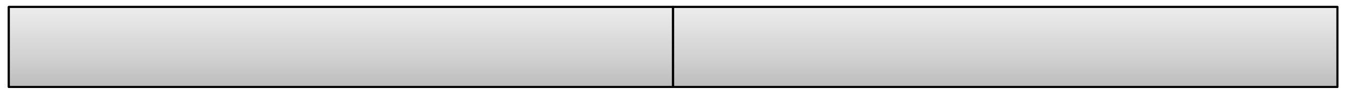
Give an algorithm that solves the problem in time at most $O(n \log n)$. It suffices to describe the algorithm in words, we do not expect detailed pseudo-code. Note, however, that you have to show that your algorithm has running time at most $O(n \log n)$.



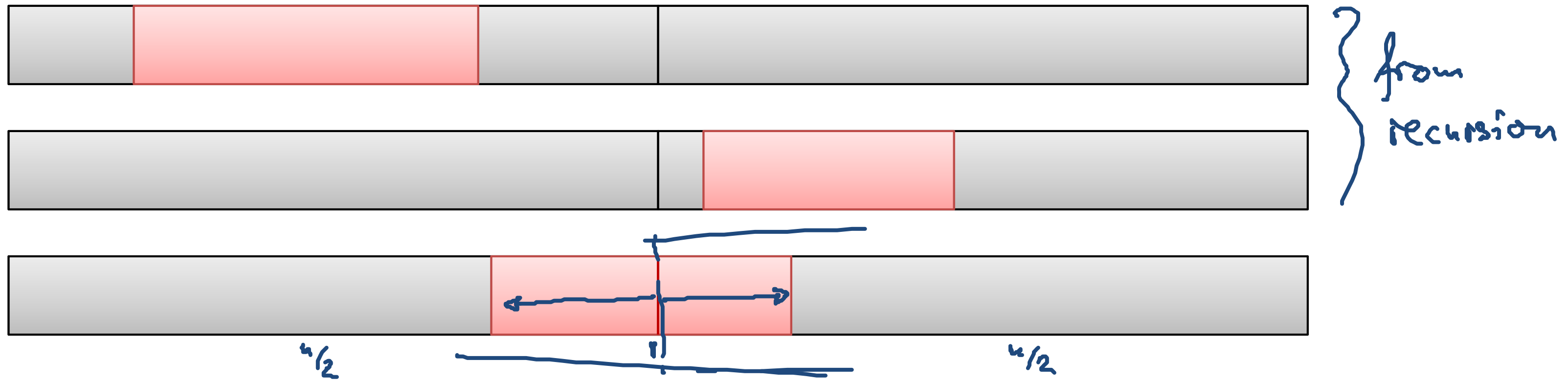
P1: Maximum Subarray Sum (18pt)

Divide & Conquer

- Split array A in two parts (left & right):



- Optimal subinterval can have 3 forms:



- Cases 1 & 2: get from recursive calls on left and right half
- Case 3:
  - Find best subarrays of left/right part starting at right/left end of part
  - Only need to check $O(n)$ cases $\Longrightarrow$ time $\underline{O(n)}$

P1: Maximum Subarray Sum (18pt)

Divide & Conquer Analysis

- Array A of length n

1. Divide into two parts of length $n/2$.
2. Solve recursively on the two parts
3. Find optimal interval for Case 3 in $O(n)$ time $\leftarrow$

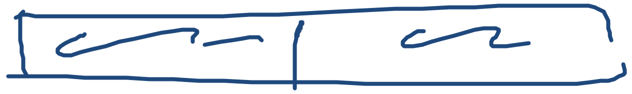
- Recurrence relation: $T(n) = 2T(n/2) + O(n)$

- Same recurrence relation as merge sort, closest pair of points, number of inversions, …

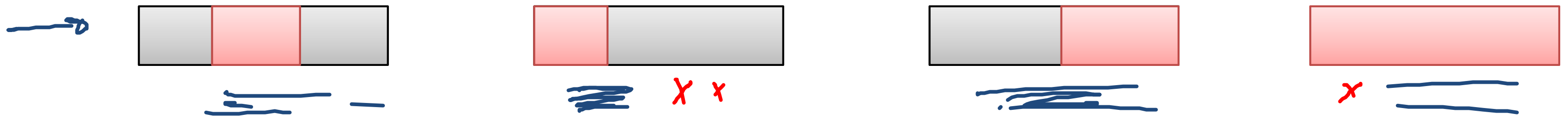
$$\Longrightarrow \text{time: } \underline{T(n) = O(n \log n)}$$

P1: Maximum Subarray Sum (18pt)

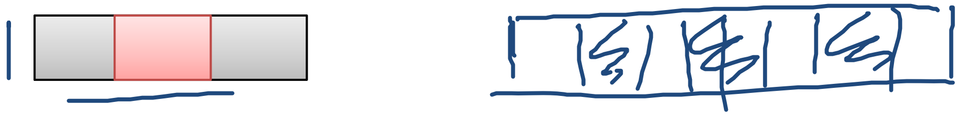
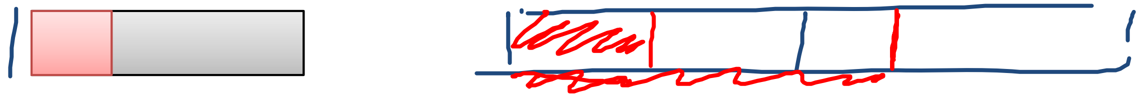
Fast Divide & Conquer



- Recursively compute best intervals of the following 4 forms:



- Combine in $O(1)$ time:

1. 
2. 
3.

4.

P1: Maximum Subarray Sum (18pt)

Fast Divide & Conquer Analysis

- Array A of length n

1. Divide into two parts of length $n/2$.
2. Solve all 4 cases recursively on the two parts
3. Find optimal intervals for all 4 cases in time $\underline{O(1)}$

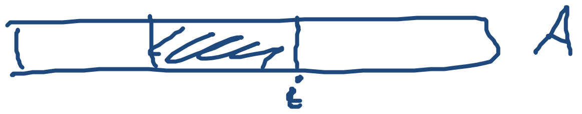
- Recurrence relation:

$$\begin{aligned} \underline{T(n) \leq 2T(n/2) +} \enclose{circle}{c} & \\ &= 4T(n/4) + 2c + c \\ &= 8T(n/8) + 4c + 2c + c \\ &\leq n \cdot T(1) + n \cdot c \end{aligned}$$

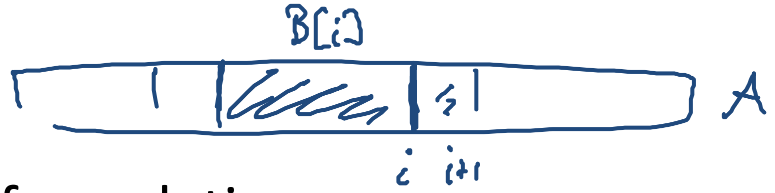
$$\Longrightarrow \text{time: } T(n) = \underline{\underline{O(n)}}$$

P1: Maximum Subarray Sum (18pt)

Dynamic Programming Solution



- For all $i \in \{1, \dots, n\}$: $\underline{B[i]}$: value of best interval ending at $A[i]$



- Recursive formulation:

$$B[i] = \max\{\underline{A[i]}, \underline{B[i-1] + A[i]}\} \leftarrow$$

$\uparrow$

B[1], B[2], ...

- Compute all $B[i]$ in time $O(n)$
  - sum of optimal interval is $\max_i B[i]$

- Get indexes of best interval in the obvious way
  - Store for each $B[i]$ also where the best interval ending at i starts

P2: Exponentiating Polynomials (13pt)

Given a polynomial $p(x)$ of degree n and an integer $k \geq 2$, the goal of this problem is to compute the k^{th} power $p^k(x)$ of $p(x)$ in an efficient way. For simplicity, we assume that k is a power of 2, that is, $k = 2^\ell$ for some integer $\ell \geq 1$. Give an efficient algorithm to compute $p^k(x)$ and analyze the running time of your algorithm!

$(p(x))^k$

Simplest solution:

1. Convert $p(x)$ into point-value repr. using the DFT alg.
   - Gives $p(\omega_N^i)$ for all $i = 0, \dots, N-1$ (and appropriate N)
2. Compute $p^k(x)$ in point-value repr.:

$$p^k(\omega_N^i) = \left(p(\omega_N^i)\right)^k$$

3. Convert back to coefficient representation using the inverse DFT alg.

Time: $O(N \cdot \log N)$ … but what is N?

P2: Exponentiating Polynomials (13pt)

Given a polynomial $p(x)$ of degree n and an integer $k \geq 2$, the goal of this problem is to compute the k^{th} power $p^k(x)$ of $p(x)$ in an efficient way. For simplicity, we assume that k is a power of 2, that is, $k = 2^\ell$ for some integer $\ell \geq 1$. Give an efficient algorithm to compute $p^k(x)$ and analyze the running time of your algorithm!

Time: $O(N \cdot \log N)$ … but what is N?

- If the degree of the polynomial $p^k(x)$ is d, N has to be at least $d + 1$
- Degree of $p^k(x)$ is $k \cdot n$
  - Hence: $N \geq kn + 1$

$$p(x) = (a_n x^n + \ldots)^k$$
$$= (a_n^k x^{kn} + \ldots)$$

- Time: $\boldsymbol{O(kn \cdot \log(kn))}$

P2: Exponentiating Polynomials (13pt)

Given a polynomial $p(x)$ of degree n and an integer $k \geq 2$, the goal of this problem is to compute the k^{th} power $p^k(x)$ of $p(x)$ in an efficient way. For simplicity, we assume that k is a power of 2, that is, $k = 2^\ell$ for some integer $\ell \geq 1$. Give an efficient algorithm to compute $p^k(x)$ and analyze the running time of your algorithm!

$p \cdot q \quad O(d \log d)$

Alternative Solution

- Compute $p^k(x) = \mathrm{p}^{2^\ell}(x)$ as

$$p^{2^\ell}(x) = \left(\left(\left(\underline{p(x)}\right)^2\right)^2 \cdots\right)^2 \Big\} \underline{\ell \text{ times}}$$

- Square the polynomial $\ell = \underline{\log k}$ times
- Degree after squaring t times: $\underline{\underline{n \cdot 2^t}}$
- Time for $\underline{\underline{t^{th}}}$ squaring:

$$\underline{O(n \cdot 2^t \cdot \log(n\overset{\uparrow}{2^t}))} \overset{\leq}{=} \underline{\underline{2^t}} \cdot \underline{O(n \cdot \log(n\underline{\underline{k}}))}$$

P2: Exponentiating Polynomials (13pt)

Given a polynomial $p(x)$ of degree n and an integer $k \geq 2$, the goal of this problem is to compute the k^{th} power $p^k(x)$ of $p(x)$ in an efficient way. For simplicity, we assume that k is a power of 2, that is, $k = 2^\ell$ for some integer $\ell \geq 1$. Give an efficient algorithm to compute $p^k(x)$ and analyze the running time of your algorithm!

Alternative Solution

- Square the polynomial $\underline{\ell} = \log k$ times
- Time for t^{th} squaring:

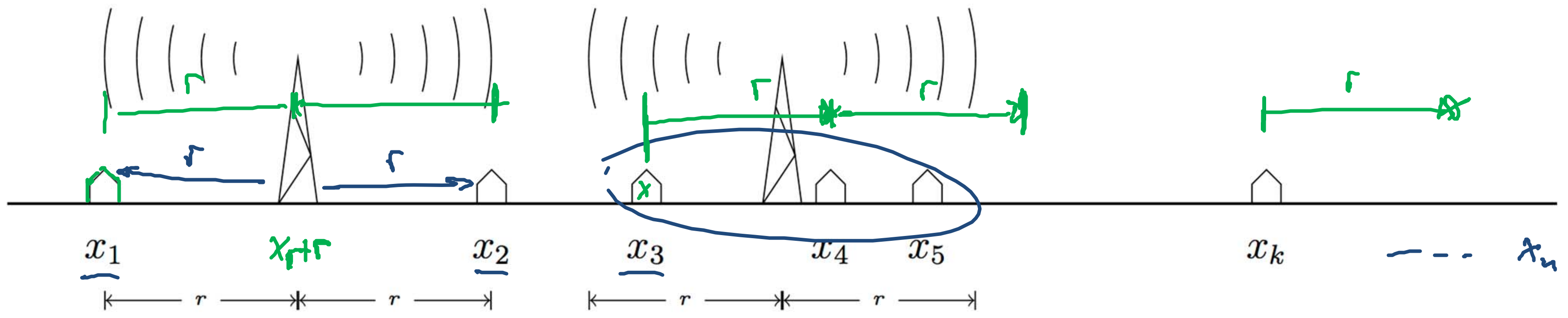
$$O(n \cdot 2^t \cdot \log(n2^t)) = \underline{2^t} \cdot \underline{O(n \cdot \log(nk))} \leftarrow$$

- Total time:

$$\boldsymbol{O(n \cdot \log(nk)) \cdot \sum_{t=1}^{\ell} 2^t = \underline{O(nk \cdot \log(nk))}}$$

$$2^{\ell+1} - 1 = O(2^\ell) = O(k)$$

P3: Placing Radio Signal Antennas (27 pt)

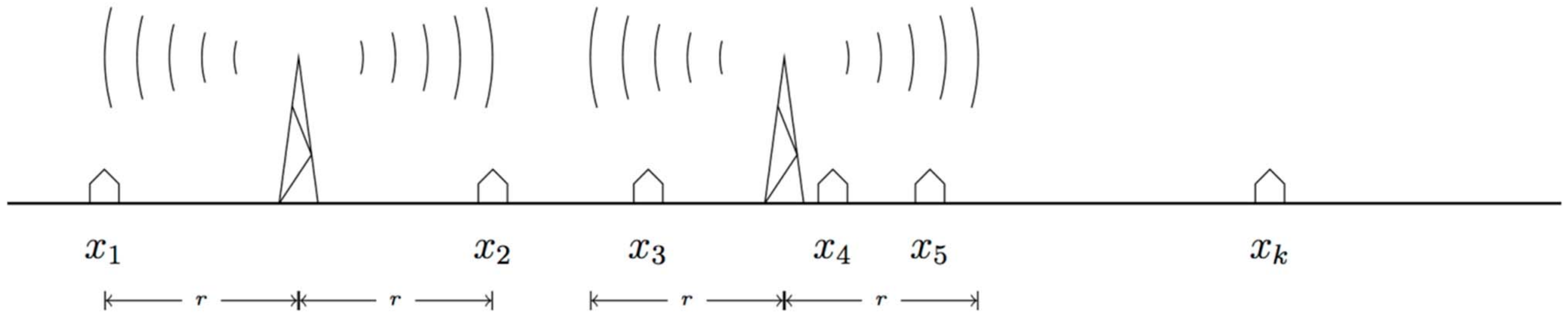


The evil villain Gamma-Zone plans to brainwash all other people in his universe, which happens to be a 1-dimensional line. The people-to-suppress have houses at positions $x_1 < x_2 < \ldots < x_n$. To reach every one of them, Gamma-Zone needs to place costly radio antennas $a_1, a_2, \ldots$, where each antenna covers all houses in a radius of r (Gamma-Zone can also place an antenna on top of a house).

(a) **(6 points)** Help Gamma-Zone by stating a (fast) greedy algorithm that calculates where to place his antennas such that he uses as few as possible while covering all houses. What is the running time of your algorithm?

(b) **(9 points)** Prove that your algorithm always computes an optimal solution.

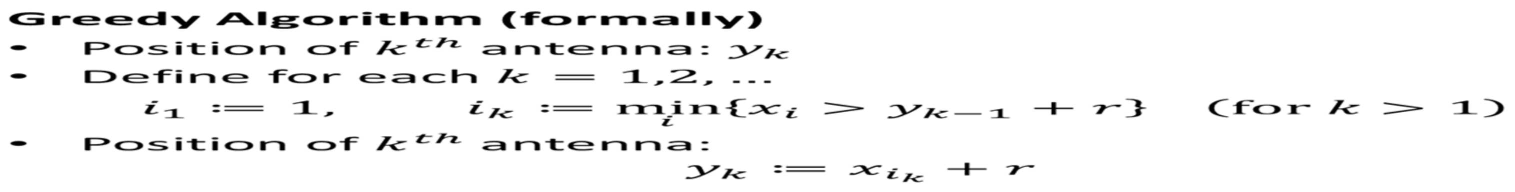
P3: Placing Radio Signal Antennas (27 pt)



Greedy Algorithm

- Place antennas as far to the right as possible:
  - First antenna at position $x_1 + r$ (first house needs to be covered)
  - Let i be the first house that's not covered by the first antenna
  - Place antenna at position $x_i + r$
  - etc.

P3: Placing Radio Signal Antennas (27 pt)



Greedy Algorithm (formally)

- Position of k^{th} antenna: y_k
- Define for each $k = 1,2,\ldots$

$$i_1 := 1, \qquad i_k := \min_i\{x_i > y_{k-1} + r\} \quad (\text{for } k > 1)$$

- Position of k^{th} antenna:

$$y_k := x_{i_k} + r$$

Greedy Algorithm (formally)

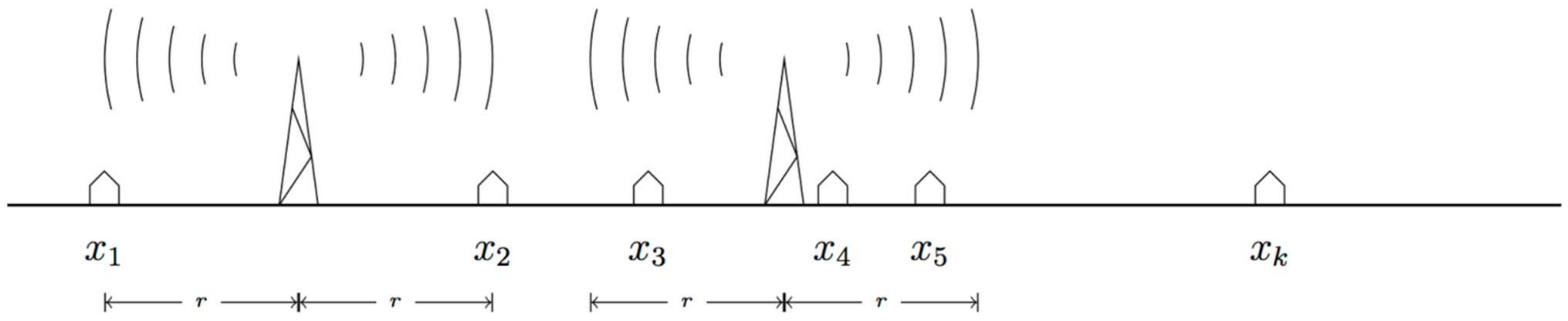
- Position of k^{th} antenna: $\underline{y_k}$
- Define for each $k = 1,2,\ldots$

$$\underline{i_1} := 1, \qquad \underline{i_k} := \min_i\{\underline{x_i} > \underline{y_{k-1} + r}\} \quad (\text{for } k > 1)$$

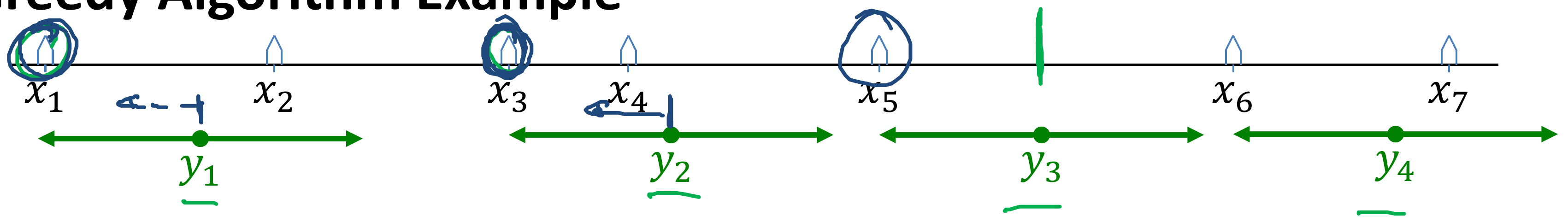
- Position of k^{th} antenna:

$$\underline{y_k := x_{i_k} + r}$$

P3: Placing Radio Signal Antennas (27 pt)



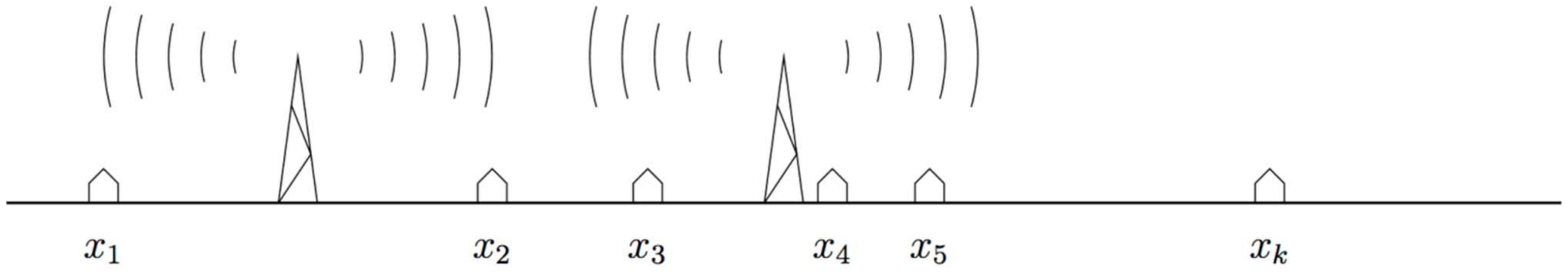
Greedy Algorithm Example



Optimality

- *Show*: In any solution, k^{th} antenna has position $\leq y_k$
- Follows by induction because
  - Induction hypothesis: first $k-1$ antennas do not cover house at x_{i_k}
  - House at x_{i_k} needs to be covered! $\leq x_{i_k} + r$

P3: Placing Radio Signal Antennas (27 pt)



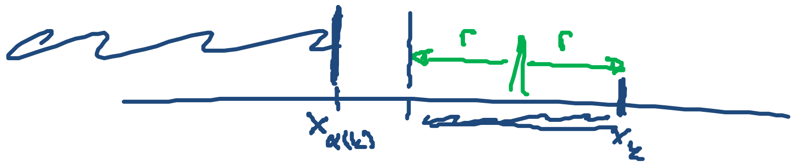
(c) **(12 points)** Realizing the opportunity, the Evil Villain Supply Company (EVSP) decides to offer two different kinds of antennas to Gamma-Zone: One costs c gold coins and has a reach of r, while the other costs $\hat{c} = 2c$ gold coins, but provides a reach of $\hat{r} = 3r$. Can you state a (fast) algorithm that minimizes Gamma-Zone's expenses? What is the running time of your algorithm?

- Greedy algorithm does not work any more
- Use dynamic programming!
- Observation:
  - can shift last antenna such that its range ends at position x_n (last house)
  - Can shift every antenna to the left such that its coverage ends at a house

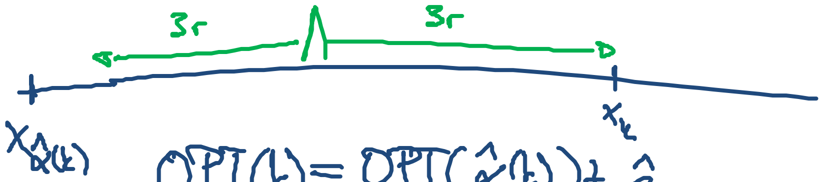
P3: Placing Radio Signal Antennas (27 pt)

(c) **(12 points)** Realizing the opportunity, the Evil Villain Supply Company (EVSP) decides to offer two different kinds of antennas to Gamma-Zone: One costs c gold coins and has a reach of r, while the other costs $\hat{c} = 2c$ gold coins, but provides a reach of $\hat{r} = 3r$. Can you state a (fast) algorithm that minimizes Gamma-Zone's expenses? What is the running time of your algorithm?

- Observation:
  - can shift last antenna such that its range ends at position x_n (last house)
  - Can shift every antenna to the left such that its coverage ends at a house
- Define $\boxed{OPT(k)}$: opt. cost to cover $x_1, \dots, x_k$
  - Can assume that coverage of last antenna ends at x_k ⟵ $OPT(k) = OPT(\alpha(k)) + c$
  - Two cases:
    1. Last antenna has reach r:



    2. Last antenna has reach $3r$:



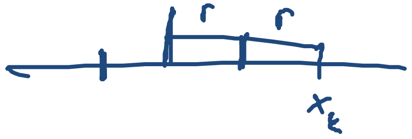
$OPT(k) = OPT(\hat{\alpha}(k)) + \underbrace{\hat{c}}_{2c}$

P3: Placing Radio Signal Antennas (27 pt)

(c) **(12 points)** Realizing the opportunity, the Evil Villain Supply Company (EVSP) decides to offer two different kinds of antennas to Gamma-Zone: One costs c gold coins and has a reach of r, while the other costs $\hat{c} = 2c$ gold coins, but provides a reach of $\hat{r} = 3r$. Can you state a (fast) algorithm that minimizes Gamma-Zone's expenses? What is the running time of your algorithm?

- $OPT(k)$: opt. cost to cover $x_1, \dots, x_k$
- Define functions $\alpha(k)$ and $\hat{\alpha}(k)$

$$\alpha(k) \coloneqq \max_i \{x_i < x_k - 2r\}$$
$$\hat{\alpha}(k) \coloneqq \max_i \{x_i < x_k - 6r\}$$



$\alpha(k) = 0$ if $x_k - 2r \le x_1$

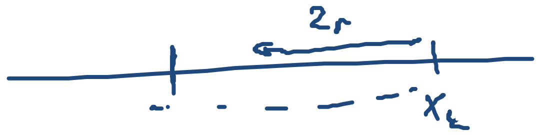
- Recursive definition for $OPT(k)$

$$\mathbf{OPT(0) = 0,}$$
$$\mathbf{OPT(k) = \min\{OPT(\alpha(k)) + c, OPT(\hat{\alpha}(k)) + \hat{c}\}}$$

P3: Placing Radio Signal Antennas (27 pt)

(c) **(12 points)** Realizing the opportunity, the Evil Villain Supply Company (EVSP) decides to offer two different kinds of antennas to Gamma-Zone: One costs c gold coins and has a reach of r, while the other costs $\hat{c} = 2c$ gold coins, but provides a reach of $\hat{r} = 3r$. Can you state a (fast) algorithm that minimizes Gamma-Zone's expenses? What is the running time of your algorithm?

- Recursive definition for $OPT(k)$



$$\boldsymbol{OPT(0) = 0,}$$

$$\boldsymbol{OPT(k) = \min\{OPT(\alpha(k)) + c, OPT(\hat{\alpha}(k)) + \hat{c}\}}$$

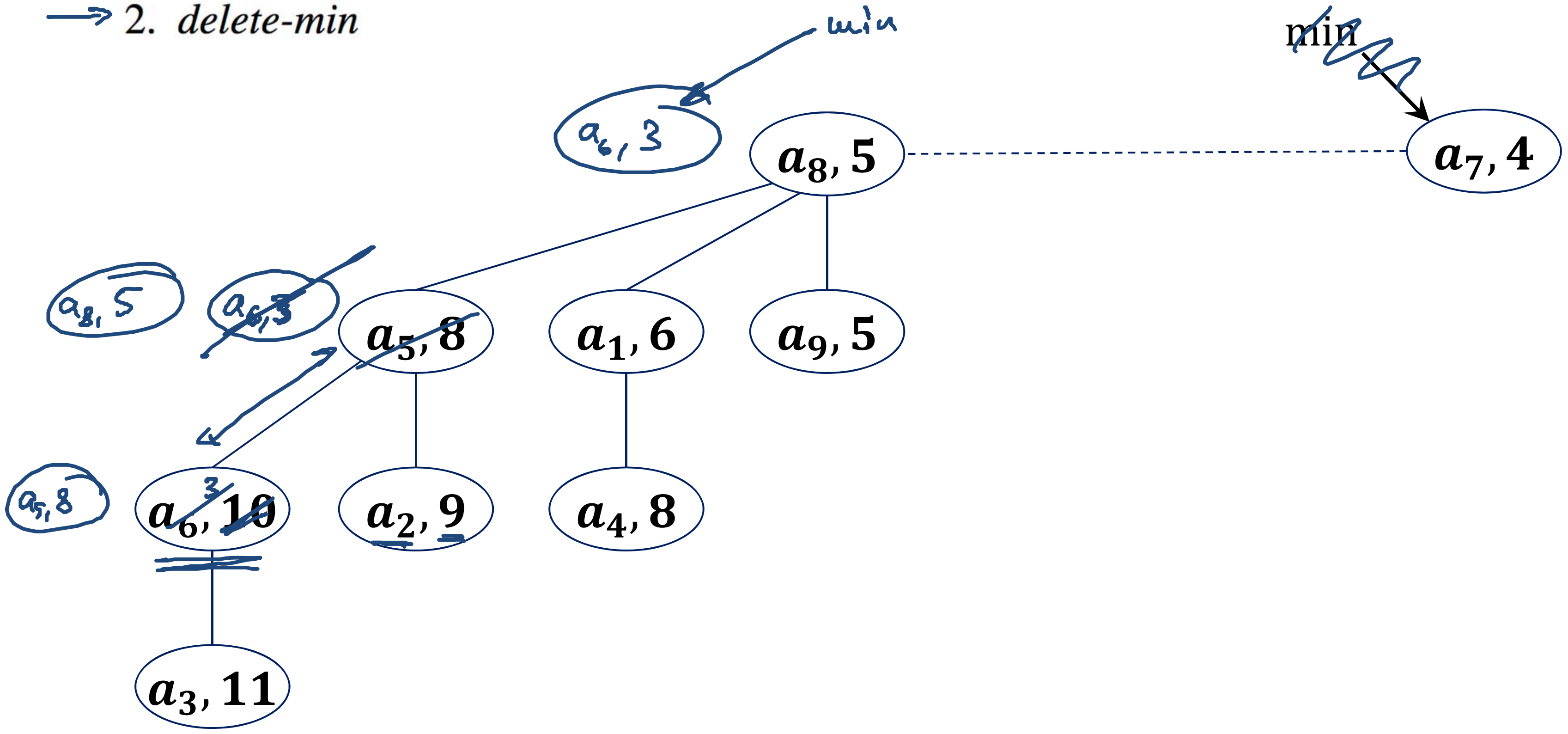
Dynamic programming algorithm

- Compute $OPT(k)$ for all $k = 1, \dots, n$ (in that order)
- Need to have $\alpha(k)$ and $\hat{\alpha}(k)$
  - Binary search: $O(\log n)$ time for each $\alpha(k)$ and $\hat{\alpha}(k)$
  - All $\alpha(k)$ and $\hat{\alpha}(k)$ can also be computed in time $O(n)$
  - Left as an exercise ;-)

P4: Binomial Heaps (18pt)

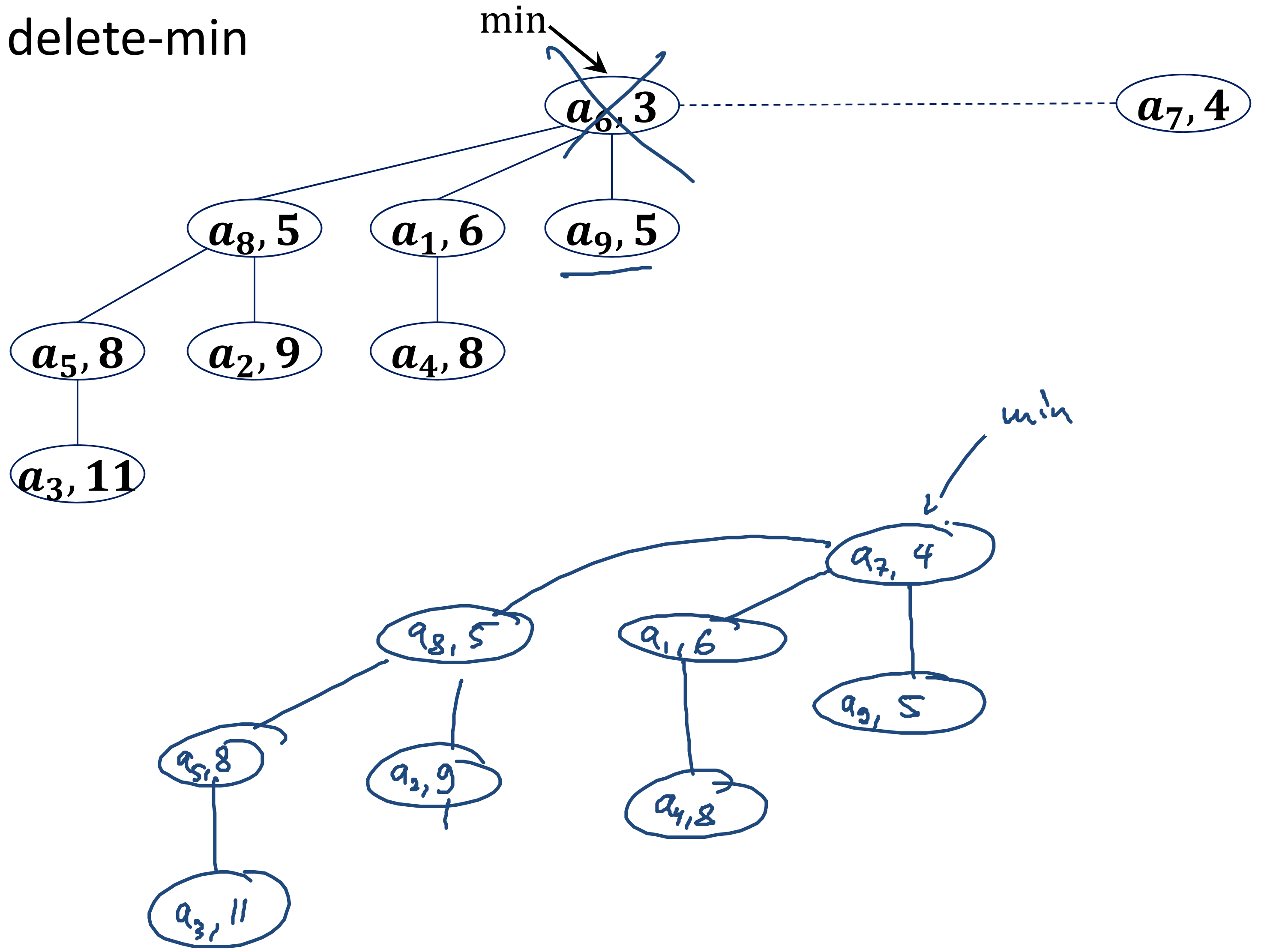
(a) **(6 points)** First, consider the following binomial heap (the values a_i represent the items, the values next to the a_i are the keys). Perform the following two operations and draw the heap after each of the operations:

1. *decrease-key*$(a_6, 3)$ (i.e., the new key of a_6 should be 3)
2. *delete-min*



P4: Binomial Heaps (18pt)

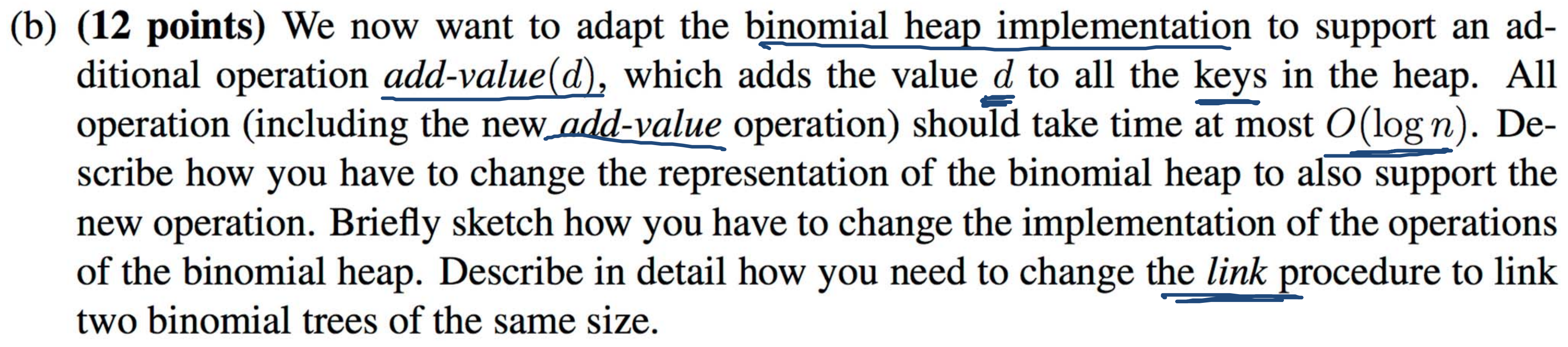
- delete-min



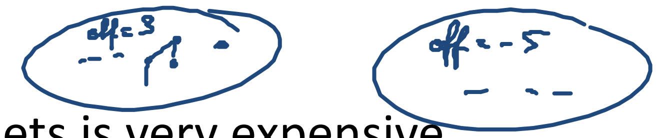
P4: Binomial Heaps (18pt)

delete-min
get-min
insert
decr.-key

merge


(b) **(12 points)** We now want to adapt the binomial heap implementation to support an additional operation *add-value*(d), which adds the value d to all the keys in the heap. All operation (including the new *add-value* operation) should take time at most $O(\log n)$. Describe how you have to change the representation of the binomial heap to also support the new operation. Briefly sketch how you have to change the implementation of the operations of the binomial heap. Describe in detail how you need to change the *link* procedure to link two binomial trees of the same size.

- Idea: just have a global variable α with an offset for all keys
- Problem: merge operation!
  - Merging two heaps with different offsets is very expensive



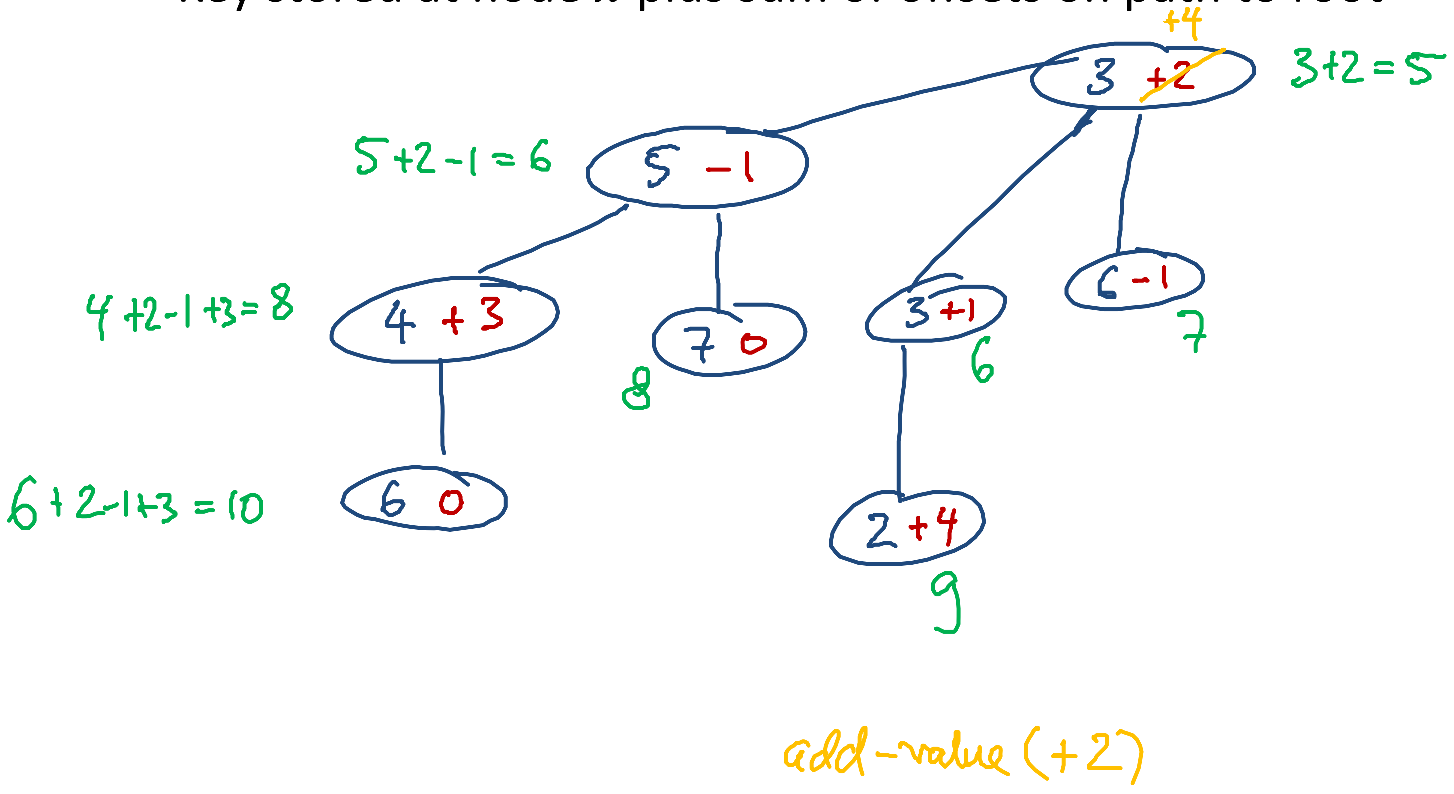
- Change idea: have an offset for every node
- Problem: *add-value* operation becomes very expensive

- Combine ideas: Have an offset for every subtree (stored at root)
  - One value per node, but
  - Changing offset for all nodes: just change root list ($O(\log n)$ nodes)

P4: Binomial Heaps (18pt)

Solution: Offset for each subtree

- Key of a node x can be obtained by:
  - Key stored at node x plus sum of offsets on path to root

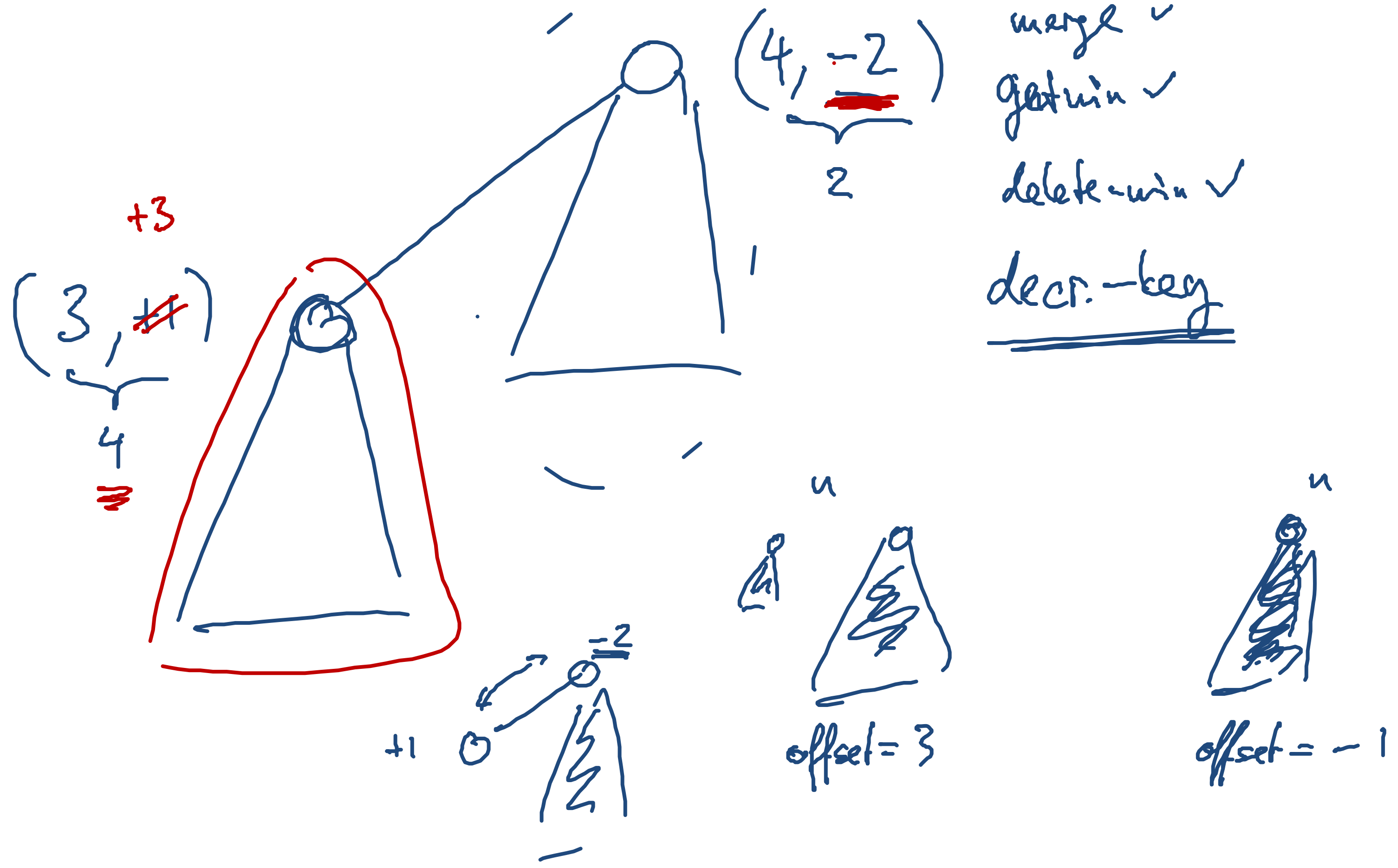


P4: Binomial Heaps (18pt)

Solution: Offset for each subtree

- Link operation

insert ✓
merge ✓
getmin ✓
delete-min ✓
decr.-key

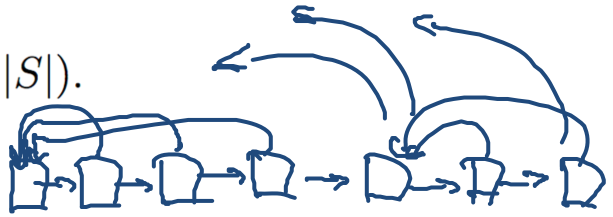


P5: Amortized Analysis (14pt)

In the lecture, we discussed the linked list implementation of the union-find data structure. Let n be the total number of *makeset* operations. We showed that if in each *union* operation the shorter list is attached to the longer list, the total cost of all (at most $n-1$) *union* operations is $\underline{O(n \log n)}$. This implies that we have time $O(1)$ for each operation and additionally $O(\log n)$ for each element (or each *makeset* operation). Hence, we get that the amortized cost of *makeset* is $O(\log n)$, whereas the amortized costs of *union* and find are $O(1)$. Because in the linked list implementation, the *union* operation is the expensive one, it would be more natural to say that the *union* operation has amortized cost $O(\log n)$ and that the other two operations have $O(1)$ amortized cost. However, the analysis from the lecture does not reflect this.

The goal here is to prove this intuition. Let $\mathcal{S}(t)$ denote the collection of sets represented by the data structure after t operations. Consider the following potential function:

$$\boxed{\Phi(t) := \sum_{S \in \mathcal{S}(t) : |S| \geq 2} |S| \left(\log_2 n - \log_2 |S| \right).}$$



(a) **(2 points)** Show that $\Phi(t)$ is a valid potential function!

(b) **(12 points)** By using the potential function $\Phi(t)$, show that the amortized costs of *makeset* and *find* are $O(1)$ and that the amortized cost of *union* is $O(\log n)$. You can assume that the actual costs of *makeset* and find are at most 1 and that the actual cost of *union* of two sets is at most the size of the smaller of the two sets.

P5: Amortized Analysis (14pt) $|S| \leq n$

$$\Phi(t) := \sum_{S \in \mathcal{S}(t): |S| \geq 2} |S| \underbrace{(\log_2 n - \log_2 |S|)}_{\geq 0}$$

(a) **(2 points)** Show that $\Phi(t)$ is a valid potential function!

(b) **(12 points)** By using the potential function $\Phi(t)$, show that the amortized costs of *makeset* and *find* are $O(1)$ and that the amortized cost of *union* is $O(\log n)$. You can assume that the actual costs of *makeset* and find are at most 1 and that the actual cost of *union* of two sets is at most the size of the smaller of the two sets.

(a) $\Phi(t) \geq 0$ ✓

(b) operation t : actual cost c_t

amortized cost a_t

$$a_t := c_t + \Phi(t) - \Phi(t-1)$$

makeset:

$$\left.\begin{array}{l} c_t \leq 1 \\ \Phi(t) = \Phi(t-1) \end{array}\right\} a_t \leq 1$$

find:

$$\left.\begin{array}{l} c_t \leq 1 \\ \Phi(t) = \Phi(t-1) \end{array}\right\} a_t \leq 1$$

P5: Amortized Analysis (14pt)

$$\Phi(t) := \sum_{S \in \mathcal{S}(t) : |S| \geq 2} |S|(\log_2 n - \log_2 |S|)$$

(a) **(2 points)** Show that $\Phi(t)$ is a valid potential function!

(b) **(12 points)** By using the potential function $\Phi(t)$, show that the amortized costs of *makeset* and *find* are $O(1)$ and that the amortized cost of *union* is $O(\log n)$. You can assume that the actual costs of *makeset* and find are at most 1 and that the actual cost of *union* of two sets is at most the size of the smaller of the two sets.

union of sets S_1 & S_2, $|S_1| \leq |S_2|$

$\underline{c_t \leq |S_1|}$ $\underline{\underline{\text{case I}}}$ $\underline{(|S_1| \geq 2)}$

$$\underline{\Phi(t) - \Phi(t-1)} = (|S_1| + |S_2|)(\log n - \log(|S_1| + |S_2|))$$
$$- \Big(|S_1|(\log n - \log|S_1|) + |S_2|(\log n - \log|S_2|)\Big)$$
$$= |S_1| \log|S_1| + |S_2| \log|S_2| - (|S_1| + |S_2|) \log(|S_1| + |S_2|)$$

P5: Amortized Analysis (14pt)

$$\Phi(t) := \sum_{S\in\mathcal{S}(t):|S|\geq 2} |S|(\log_2 n - \log_2|S|)$$

(a) **(2 points)** Show that $\Phi(t)$ is a valid potential function!

$\underline{(|S_1| \leq |S_2|)}$

(b) **(12 points)** By using the potential function $\Phi(t)$, show that the amortized costs of *makeset* and *find* are $O(1)$ and that the amortized cost of *union* is $O(\log n)$. You can assume that the actual costs of *makeset* and find are at most 1 and that the actual cost of *union* of two sets is at most the size of the smaller of the two sets.

$$a_t = |S_1| + |S_1|\log|S_1| + |S_2|\log|S_2| - \underline{(|S_1|+|S_2|)(\log(|S_1|+|S_2|)}$$

$$= |S_1|(\log|S_1|+1) - |S_1|\underbrace{\log(|S_1|+|S_2|)}_{\substack{\geq \log(2|S_1|) \\ = \log|S_1|+1}} + |S_2|\underbrace{(\log|S_2| - \log(|S_1|+|S_2|))}_{\leq 0}$$

$$\underline{\leq \quad 0}$$

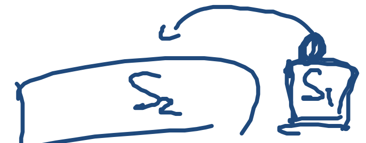
P5: Amortized Analysis (14pt)

$$\Phi(t) \coloneqq \sum_{S\in\mathcal{S}(t):|S|\geq 2} \underbrace{|S|}_{\uparrow\ 2(\log n)}(\log_2 n - \log_2|S|)$$

(a) **(2 points)** Show that $\Phi(t)$ is a valid potential function!

(b) **(12 points)** By using the potential function $\Phi(t)$, show that the amortized costs of *makeset* and *find* are $O(1)$ and that the amortized cost of *union* is $O(\log n)$. You can assume that the actual costs of *makeset* and find are at most 1 and that the actual cost of *union* of two sets is at most the size of the smaller of the two sets.

case II: $|S_1| = 1$



$|S_2| \geq 2$

$c_t \leq 1$

$$\phi(t) - \phi(t-1) = (|S_2|+1)(\log n - \log(|S_2|+1)) - |S_2|(\log n - \log|S_2|)$$

$$= \log n + \underbrace{|S_2|\log|S_2| - (|S_2|+1)\log(|S_2|+1)}_{\leq 0}$$

$$\leq \log n$$