Amortized analysis

Peter Hajnal

Bolyai Institute, University of Szeged, Hungary

2023 fall

Worst case analysis vs amortized analysis

Worst case analysis vs amortized analysis

Analysis of the algorithms studied so far were based on a worst case analysis: We ran the algorithm on an input and the runtime was bounded by a function of the length/size of the input.

Worst case analysis vs amortized analysis

Analysis of the algorithms studied so far were based on a worst case analysis: We ran the algorithm on an input and the runtime was bounded by a function of the length/size of the input.

Many times our algorithm is a repetition of complex procedures several times.

Worst case analysis vs amortized analysis

Analysis of the algorithms studied so far were based on a worst case analysis: We ran the algorithm on an input and the runtime was bounded by a function of the length/size of the input.

Many times our algorithm is a repetition of complex procedures several times. We can bound the complexity of the algorithm with the product of the number of repetitions, and the worst case analysis of the procedure.

Worst case analysis vs amortized analysis

Analysis of the algorithms studied so far were based on a worst case analysis: We ran the algorithm on an input and the runtime was bounded by a function of the length/size of the input.

Many times our algorithm is a repetition of complex procedures several times. We can bound the complexity of the algorithm with the product of the number of repetitions, and the worst case analysis of the procedure.

The above bound is often very loose.

Worst case analysis vs amortized analysis

Analysis of the algorithms studied so far were based on a worst case analysis: We ran the algorithm on an input and the runtime was bounded by a function of the length/size of the input.

Many times our algorithm is a repetition of complex procedures several times. We can bound the complexity of the algorithm with the product of the number of repetitions, and the worst case analysis of the procedure.

The above bound is often very loose. If the algorithm executes many times an operation, it is possible that once to perform tha operation takes many steps, while at another time the cost is significantly less.

Worst case analysis vs amortized analysis

Analysis of the algorithms studied so far were based on a worst case analysis: We ran the algorithm on an input and the runtime was bounded by a function of the length/size of the input.

Many times our algorithm is a repetition of complex procedures several times. We can bound the complexity of the algorithm with the product of the number of repetitions, and the worst case analysis of the procedure.

The above bound is often very loose. If the algorithm executes many times an operation, it is possible that once to perform tha operation takes many steps, while at another time the cost is significantly less.

If our analysis of the repeated operations takes these final remarks into the account we say that we did an amortized analysis. Again the most important thing is to see and understand specific examples.

An introductory example: Binary counter

An introductory example: Binary counter

The input

Given $0^n = (0, 0, \dots, 0)$, a string of n 0's.

An introductory example: Binary counter

The input

Given $0^n = (0, 0, \ldots, 0)$, a string of n 0's.

It can also be considered as the number 0 in binary notation using n digits.

An introductory example: Binary counter

The input

Given $0^n = (0, 0, \ldots, 0)$, a string of n 0's.

It can also be considered as the number 0 in binary notation using n digits.

The task

Increase the actual number by 1 and 'print' the corresponding n digits to the top of the actual string.

An introductory example: Binary counter

The input

Given $0^n = (0, 0, \ldots, 0)$, a string of n 0's.

It can also be considered as the number 0 in binary notation using n digits.

The task

Increase the actual number by 1 and 'print' the corresponding n digits to the top of the actual string. After $2^n - 1$ updates we stop with 1^n.

An introductory example: Binary counter

The input

Given $0^n = (0, 0, \ldots, 0)$, a string of n 0's.

It can also be considered as the number 0 in binary notation using n digits.

The task

Increase the actual number by 1 and 'print' the corresponding n digits to the top of the actual string. After $2^n - 1$ updates we stop with 1^n.

The cost

The cost of an update will be the number of bits that are changed.

An introductory example: Binary counter

The input

Given $0^n = (0, 0, \ldots, 0)$, a string of n 0's.

It can also be considered as the number 0 in binary notation using n digits.

The task

Increase the actual number by 1 and 'print' the corresponding n digits to the top of the actual string. After $2^n - 1$ updates we stop with 1^n.

The cost

The cost of an update will be the number of bits that are changed.

What is the total cost of the entire algorithm?

An introductory example: Binary counter

The input

Given $0^n = (0, 0, \ldots, 0)$, a string of n 0's.

It can also be considered as the number 0 in binary notation using n digits.

The task

Increase the actual number by 1 and 'print' the corresponding n digits to the top of the actual string. After $2^n - 1$ updates we stop with 1^n.

The cost

The cost of an update will be the number of bits that are changed.

What is the total cost of the entire algorithm? The task as a computational problem is not very interesting, however the analysis will be instructive.

The logic of worst case analysis

The logic of worst case analysis

If the actual string is $01^{n-1} = 011\ldots11$, then after the update we will see $10^{n-1} = 100\ldots00$.

The logic of worst case analysis

If the actual string is $01^{n-1} = 011\ldots11$, then after the update we will see $10^{n-1} = 100\ldots00$.

This special update costs n, that is the highest cost during the algorithm.

The logic of worst case analysis

If the actual string is $01^{n-1} = 011\ldots11$, then after the update we will see $10^{n-1} = 100\ldots00$.

This special update costs n, that is the highest cost during the algorithm.

Thus, the cost of the $2^n - 1$ updates can be estimated by $(2^n - 1)n$.

Some important remarks

Some important remarks

Notice that during each update the final 1-block of our bit string is converted to a block of 0's and the previous single 0 bit is rewritten to 1.

Some important remarks

Notice that during each update the final 1-block of our bit string is converted to a block of 0's and the previous single 0 bit is rewritten to 1.

If the bit sequence ends with 0, then final 1-block is empty.

Some important remarks

Notice that during each update the final 1-block of our bit string is converted to a block of 0's and the previous single 0 bit is rewritten to 1.

If the bit sequence ends with 0, then final 1-block is empty. If the closing 1 block is the entire string, then there is no update, we stop.

Some important remarks

Notice that during each update the final 1-block of our bit string is converted to a block of 0's and the previous single 0 bit is rewritten to 1.

If the bit sequence ends with 0, then final 1-block is empty. If the closing 1 block is the entire string, then there is no update, we stop.

If the cost is c, then there is a single $0 \leftarrow 1$ rewrite and $c - 1$ $1 \leftarrow 0$ rewrite's.

The idea of amortized analysis (binary counter problem)

The idea of amortized analysis (binary counter problem)

Suppose if we change a bit we have to pay \$1.

The idea of amortized analysis (binary counter problem)

Suppose if we change a bit we have to pay \$1.

If we were to prepare for the worst case scenario, then we have to take \$$n$ with us to perform an update.

The idea of amortized analysis (binary counter problem)

Suppose if we change a bit we have to pay \$1.

If we were to prepare for the worst case scenario, then we have to take \$$n$ with us to perform an update.

However, if we have a good economic sense, then we'll make it with \$2!

The idea of amortized analysis (binary counter problem)

Suppose if we change a bit we have to pay \$1.

If we were to prepare for the worst case scenario, then we have to take \$$n$ with us to perform an update.

However, if we have a good economic sense, then we'll make it with \$2!

The idea

If we rewrite a 0 to 1 ($0 \leftarrow 1$), then we have \$1 to pay for it. We have one leftover \$1. We can deposit it at the rewritten 1.

The idea of amortized analysis (binary counter problem)

Suppose if we change a bit we have to pay \$1.

If we were to prepare for the worst case scenario, then we have to take \$$n$ with us to perform an update.

However, if we have a good economic sense, then we'll make it with \$2!

The idea

If we rewrite a 0 to 1 ($0 \leftarrow 1$), then we have \$1 to pay for it. We have one leftover \$1. We can deposit it at the rewritten 1.

Observation

It will be true that our current bit sequence contains \$1 deposit on every 1 bit (the future cost of the $1 \leftarrow 0$ rewrites).

The theorem

The theorem

The theorem

The cost of the binary counter is exactly

$$(2^n - 1)2 - n.$$

The language of amortization: Economics

The language of amortization: Economics

When we perform the procedure once we charge for that step a 'cost' component plus a 'deposit' component.

The language of amortization: Economics

When we perform the procedure once we charge for that step a 'cost' component plus a 'deposit' component.

The actual cost and some of the previous deposits must cover the computation performed.

The language of amortization: Economics

When we perform the procedure once we charge for that step a 'cost' component plus a 'deposit' component.

The actual cost and some of the previous deposits must cover the computation performed.

In our case (binary counter problem), the amortized cost of each update is \$2.

The language of amortization: Economics

When we perform the procedure once we charge for that step a 'cost' component plus a 'deposit' component.

The actual cost and some of the previous deposits must cover the computation performed.

In our case (binary counter problem), the amortized cost of each update is $2.

The above interpretation is a called *banker's interpretation*.

Another language of amortization: Physics

Another language of amortization: Physics

We define a potential for the current string:

Another language of amortization: Physics

We define a potential for the current string: The potential is just the number of 1's. Hence the initial potential is 0, the final one is 1.

Another language of amortization: Physics

We define a potential for the current string: The potential is just the number of 1's. Hence the initial potential is 0, the final one is 1.

In each step we can increase the potential, and that has a cost/price.

Another language of amortization: Physics

We define a potential for the current string: The potential is just the number of 1's. Hence the initial potential is 0, the final one is 1.

In each step we can increase the potential, and that has a cost/price.

But we can use the potential to 'gain energy'/do computation.

Another language of amortization: Physics

We define a potential for the current string: The potential is just the number of 1's. Hence the initial potential is 0, the final one is 1.

In each step we can increase the potential, and that has a cost/price.

But we can use the potential to 'gain energy'/do computation.

Formally the potential function is $P : \mathcal{K} \to \mathbb{R}$, where $\mathcal{K}$ is the set of configurations (strings in our example).

Another language of amortization: Physics

We define a potential for the current string: The potential is just the number of 1's. Hence the initial potential is 0, the final one is 1.

In each step we can increase the potential, and that has a cost/price.

But we can use the potential to 'gain energy'/do computation.

Formally the potential function is $P : \mathcal{K} \to \mathbb{R}$, where $\mathcal{K}$ is the set of configurations (strings in our example).

Definition: Amortized cost

The i-th update $\kappa_i \leftarrow \kappa_{i-1}$,

$$c_{\text{amort}}(e_i) = c(e_i) + P(\kappa_i) - P(\kappa_{i-1}).$$

Break

The fundamental problem

Definition: The convex hull problem

Given n points in the coordinate plane: (x_1, y_1), (x_2, y_2), (x_3, y_3), …, (x_n, y_n) so that their x coordinates are strictly monotonically increasing.

The fundamental problem

Definition: The convex hull problem

Given n points in the coordinate plane: (x_1, y_1), (x_2, y_2), (x_3, y_3), $\ldots$, (x_n, y_n) so that their x coordinates are strictly monotonically increasing.

Determine the convex hull of the input set (which points are the vertices of the convex hull, and how they are circularly arranged on it).

Simplification: Determining the upper hull

Definition: Upper hull

Simplification: Determining the upper hull

Definition: Upper hull

Take the point with minimal x-coordinate and the point with maximal x-coordinate.

Simplification: Determining the upper hull

Definition: Upper hull

Take the point with minimal x-coordinate and the point with maximal x-coordinate.
These two points divide the convex hull into two segments.

Simplification: Determining the upper hull

Definition: Upper hull

Take the point with minimal x-coordinate and the point with maximal x-coordinate.
These two points divide the convex hull into two segments.
The two segments are called upper hull and lower hull.

Simplification: Determining the upper hull

Definition: Upper hull

Take the point with minimal x-coordinate and the point with maximal x-coordinate.
These two points divide the convex hull into two segments.
The two segments are called upper hull and lower hull.

The upper and lower hulls play symmetrical role, and together they form the convex hull of our point set.

Simplification: Determining the upper hull

Definition: Upper hull

Take the point with minimal x-coordinate and the point with maximal x-coordinate.
These two points divide the convex hull into two segments.
The two segments are called upper hull and lower hull.

The upper and lower hulls play symmetrical role, and together they form the convex hull of our point set.

It is enough to compute the upper hull.

Dynamic programming approach

Dynamic programming approach

The set of problems

Let $\mathcal{P}_i$ be the set of the first i points. Determine their upper hull:

$$\mathcal{U}_i : E = P_1, P_{i_2}, P_{i_3}, \ldots, P_{i_{\ell-1}}, P_i.$$

Dynamic programming approach

The set of problems

Let $\mathcal{P}_i$ be the set of the first i points. Determine their upper hull:

$$\mathcal{U}_i : E = P_1, P_{i_2}, P_{i_3}, \ldots, P_{i_{\ell-1}}, P_i.$$

The order of our problems

$$\mathcal{U}_1 \to \mathcal{U}_2 \to \mathcal{U}_3 \to \ldots \to \mathcal{U}_{n-1} \to \mathcal{U}_n.$$

Dynamic programming approach

The set of problems

Let $\mathcal{P}_i$ be the set of the first i points. Determine their upper hull:

$$\mathcal{U}_i : E = P_1, P_{i_2}, P_{i_3}, \ldots, P_{i_{\ell-1}}, P_i.$$

The order of our problems

$$\mathcal{U}_1 \to \mathcal{U}_2 \to \mathcal{U}_3 \to \ldots \to \mathcal{U}_{n-1} \to \mathcal{U}_n.$$

Dynamic programming requires to solve the following problem:

Dynamic programming approach

The set of problems

Let $\mathcal{P}_i$ be the set of the first i points. Determine their upper hull:

$$\mathcal{U}_i : E = P_1, P_{i_2}, P_{i_3}, \ldots, P_{i_{\ell-1}}, P_i.$$

The order of our problems

$$\mathcal{U}_1 \to \mathcal{U}_2 \to \mathcal{U}_3 \to \ldots \to \mathcal{U}_{n-1} \to \mathcal{U}_n.$$

Dynamic programming requires to solve the following problem: Insert the point P_{i+1} into the $\mathcal{U}_i$ (the upper hall of the ith problem).

The insertion with binary search

The insertion with binary search

The new upper hull is the starting segment of the previous one extended by P_{i+1}:

The insertion with binary search

The new upper hull is the starting segment of the previous one extended by P_{i+1}:

$$\mathcal{F}_i : E = P_1, P_{i_2}, P_{i_3}, \ldots, P_{i_{k-1}}, U = P_{i_k}, P_{i+1}.$$

The insertion with binary search

The new upper hull is the starting segment of the previous one extended by P_{i+1}:

$$\mathcal{F}_i : E = P_1, P_{i_2}, P_{i_3}, \dots, P_{i_{k-1}}, U = P_{i_k}, P_{i+1}.$$

Insertion is determining the vertex U.

The insertion with binary search

The new upper hull is the starting segment of the previous one extended by P_{i+1}:

$$\mathcal{F}_i : E = P_1, P_{i_2}, P_{i_3}, \ldots, P_{i_{k-1}}, U = P_{i_k}, P_{i+1}.$$

Insertion is determining the vertex U.

The condition that defines U is:

$$m(P_{i_{k-1}} P_{i+1}) > m(UP_{i+1}) \leq m(P_{i_{k+1}} P_{i+1}),$$

The insertion with binary search

The new upper hull is the starting segment of the previous one extended by P_{i+1}:

$$\mathcal{F}_i : E = P_1, P_{i_2}, P_{i_3}, \ldots, P_{i_{k-1}}, U = P_{i_k}, P_{i+1}.$$

Insertion is determining the vertex U.

The condition that defines U is:

$$m(P_{i_{k-1}} P_{i+1}) > m(UP_{i+1}) \leq m(P_{i_{k+1}} P_{i+1}),$$

where $m(AB)$ is the slope of the line AB.

Search for U: I. binary search

Search for U: I. binary search

Claim

$\mathcal{O}(\log \ell) = \mathcal{O}(\log i)$ you can find the P_j you are looking for.

Search for U: I. binary search

Claim

$\mathcal{O}(\log \ell) = \mathcal{O}(\log i)$ you can find the P_j you are looking for.

The total cost is

$$\mathcal{O}(\log 1 + \log 2 + \ldots + \log(n-1)) = \mathcal{O}(n \log n).$$

Search for U: I. naiv search

Search for U: I. naiv search

Let's test the point P_{i_ℓ}.

Search for U: I. naiv search

Let's test the point P_{i_ℓ}. If it's not good for the role of U, let's test the point $P_{i_{\ell-1}}$.

Search for U: I. naiv search

Let's test the point P_{i_ℓ}. If it's not good for the role of U, let's test the point $P_{i_{\ell-1}}$. If it's not good for the role of U, let's test it the $P_{i_{\ell-2}}$ vertex.

Search for U: I. naiv search

Let's test the point P_{i_ℓ}. If it's not good for the role of U, let's test the point $P_{i_{\ell-1}}$. If it's not good for the role of U, let's test it the $P_{i_{\ell-2}}$ vertex. And so on.

Search for U: I. naiv search

Let's test the point P_{i_ℓ}. If it's not good for the role of U, let's test the point $P_{i_{\ell-1}}$. If it's not good for the role of U, let's test it the $P_{i_{\ell-2}}$ vertex. And so on.

Maybe surprising, but this algorithm is better than the one based on binary search.

Amortized analysis

Amortized analysis

A test costs $\mathcal{O}(1) = c$.

Amortized analysis

A test costs $\mathcal{O}(1) = c$.

The amortized cost of inserting P is $2c = c + c$.

Amortized analysis

A test costs $\mathcal{O}(1) = c$.

The amortized cost of inserting P is $2c = c + c$.

The first term is the cost of the succesful test.

Amortized analysis

A test costs $\mathcal{O}(1) = c$.

The amortized cost of inserting P is $2c = c + c$.

The first term is the cost of the succesful test. The second term is a deposit, an "exit costs" for a future 'failed test'.

Amortized analysis

A test costs $\mathcal{O}(1) = c$.

The amortized cost of inserting P is $2c = c + c$.

The first term is the cost of the succesful test. The second term is a deposit, an "exit costs" for a future 'failed test'.

Observation

If a P_j is tested for U during the insertion of P_{i+1} and it is failed, then P_j is discarded from the algorithm.

Amortized analysis

A test costs $\mathcal{O}(1) = c$.

The amortized cost of inserting P is $2c = c + c$.

The first term is the cost of the succesful test. The second term is a deposit, an "exit costs" for a future 'failed test'.

Observation

If a P_j is tested for U during the insertion of P_{i+1} and it is failed, then P_j is discarded from the algorithm.

Theorem

The total cost of determining the upper hall based on the naive searxh is

$$\mathcal{O}(n).$$

This is the end!

Thank you for your attention!